



Avec les Nuls, tout devient facile !

Les algorithmes

pour

les nuls



- La conception des algorithmes
- Utiliser Python pour la programmation algorithmique
- Les bases de la théorie des graphes
- Utiliser la programmation dynamique
- La découverte de l'heuristique

John Paul Mueller
Luca Massaron

VISIT...

LANZAROTE
Caliente.COM



Les algorithmes pour **les nuls**

John Paul Mueller et Luca Massaron

FIRST
 Editions

Les algorithmes pour les Nuls

Pour les Nuls est une marque déposée de Wiley Publishing, Inc.
For Dummies est une marque déposée de Wiley Publishing, Inc.

Collection dirigée par Jean-Pierre Cano
Traduction : Marc Rozenbaum
Mise en page : maged

Edition française publiée en accord avec Wiley Publishing, Inc.
© Éditions First, un département d'Édi8, 2017
Éditions First, un département d'Édi8
12 avenue d'Italie
75013 Paris
Tél. : 01 44 16 09 00
Fax : 01 44 16 09 01
E-mail : firstinfo@efirst.com
Web : www.editionsfirst.fr
ISBN : 978-2-412-02590-1
ISBN numérique : 9782412033036
Dépôt légal : 3^e trimestre 2017

Cette œuvre est protégée par le droit d'auteur et strictement réservée à l'usage privé du client. Toute reproduction ou diffusion au profit de tiers, à titre gratuit ou onéreux, de tout ou partie de cette œuvre est strictement interdite et constitue une contrefaçon prévue par les articles L 335-2 et suivants du Code de la propriété intellectuelle. L'éditeur se réserve le droit de poursuivre toute atteinte à ses droits de propriété intellectuelle devant les juridictions civiles ou pénales.

Ce livre numérique a été converti initialement au format EPUB par Isako www.isako.com à partir de l'édition papier du même ouvrage.

Introduction

Vous avez besoin d'acquérir des notions d'algorithmique dans le cadre de vos études, ou de votre travail. Or, les ouvrages que vous avez déjà consultés jusqu'ici sur ce sujet vous ont surtout semblé soporifiques et vous n'avez pas eu l'impression de pouvoir en tirer beaucoup de connaissances. Même si vous n'avez pas été découragé par ces symboles mystérieux qui doivent avoir été tracés par un bambin de deux ans amateur de gribouillis, vous finissez par vous demander ce que tout cela pourrait bien vous apporter. Les mathématiques, c'est généralement fastidieux ! Cependant, avec *Les Algorithmes pour les Nuls*, ce sera différent. La première chose que vous remarquerez dans ce livre est que les symboles ésotériques y sont bien rares (surtout ceux évoquant les gribouillis). Certes, vous en trouverez quelques-uns (il s'agit tout de même d'un ouvrage de mathématiques), mais ce que ce livre contient, ce sont surtout des instructions claires pour l'utilisation d'algorithmes qui ont déjà un nom et une histoire et qui servent à exécuter des tâches utiles. Vous y découvrirez des techniques de codage simples pour l'exécution de programmes qui produiront des résultats surprenants, de quoi impressionner vos amis et, à coup sûr, les rendre jaloux des prouesses que vous réaliserez avec des mathématiques échappant à leur compréhension. Et tout cela, vous allez l'obtenir sans vous casser la tête le moins du monde, et sans même commencer à bâiller (sauf si c'est délibéré de votre part).

À propos de ce livre

Les Algorithmes pour les Nuls, c'est le livre de mathématiques que vous auriez aimé avoir sous la main quand vous étiez au lycée ou à l'université. Vous allez vous rendre compte, par exemple, que les algorithmes ne datent pas d'hier. Mille six cents ans avant J.-C., les Babyloniens en utilisaient déjà pour exécuter des tâches simples. Si les Babyloniens en étaient capables, vous en êtes certainement capable vous aussi ! En fait, ce livre contient trois choses que vous ne trouverez pas dans la plupart des ouvrages de mathématiques :

- » Des algorithmes qui ont des noms et une histoire, ce qui vous permet de mieux vous en souvenir et de comprendre pourquoi quelqu'un s'est donné la peine de les inventer.
- » Des explications simples concernant la manière dont les algorithmes permettent d'obtenir des résultats remarquables en termes de manipulation de données, d'analyse de données et de calcul de probabilités.
- » Des lignes de code montrant comment utiliser l'algorithme sans être obligé de passer par des symboles mystérieux hors de portée de quiconque n'est pas titulaire d'un diplôme de maths.

Ce livre traite en particulier de l'utilisation des bons outils. Il utilise Python pour effectuer différentes tâches. Python possède des caractéristiques particulières grâce auxquelles travailler avec des algorithmes devient nettement plus facile. Ainsi, par exemple, Python vous donne accès à un vaste ensemble de modules qui vous permettent de faire quasiment tout ce que vous pouvez imaginer, et bien davantage encore. Toutefois, contrairement à beaucoup d'ouvrages consacrés à Python, celui-ci ne vous inonde pas de modules. Nous utilisons une sélection de modules qui permet une grande souplesse et offre de nombreuses fonctionnalités, mais sans que cela ne vous coûte rien. Vous allez

pouvoir parcourir ce livre en entier sans devoir déboursier un centime de l'argent que vous avez durement gagné.

Dans ce livre, vous allez aussi découvrir des techniques intéressantes. Le plus important, c'est que vous n'allez pas simplement observer les algorithmes utilisés pour exécuter des tâches : ce livre vous explique aussi comment ces algorithmes fonctionnent. À la différence de nombreux autres livres, il vous permet de bien comprendre ce que vous faites, même si vous n'êtes pas titulaire d'un doctorat en mathématiques. Dans chaque exemple, nous vous montrons le résultat attendu en vous expliquant pourquoi ce résultat est important. Ainsi, vous ne restez pas sur l'impression que quelque chose vous manque.

Naturellement, peut-être le problème de l'environnement de programmation vous préoccupe-t-il aussi, et là encore, ce livre vous apporte l'éclairage nécessaire. Toutes les instructions pour l'installation d'Anaconda, l'environnement de développement intégré du langage Python utilisé dans ce livre, vous sont données dès le début. Vous y trouverez également des précisions (accompagnées de références) qui vous permettront de comprendre la programmation de base avec Python que vous aurez besoin de réaliser. Tout est conçu pour que vous ayez le pied à l'étrier le plus rapidement possible, avec des exemples simples et pratiques, de telle sorte que le code ne devienne pas un obstacle à votre apprentissage.

Afin de mieux vous permettre d'assimiler les concepts, ce livre utilise les conventions suivantes :

- » Le texte que vous devez saisir tel qu'il apparaît dans le livre est en **gras**, sauf dans les listes d'étapes : chaque étape étant en gras, le texte à saisir n'est pas en gras.
- » Les mots ou expressions qui sont aussi en *italique* représentent des zones à remplir, ce qui signifie que vous devez les remplacer par des mots ou des expressions correspondant à votre situation. Ainsi, par exemple, s'il est écrit « Saisissez *Votre Nom* et appuyez sur la touche

Entrée », vous devrez remplacer *Votre Nom* par votre vrai nom.

- » Nous utilisons aussi l'*italique* pour les termes dont nous donnons la définition. Autrement dit, vous n'aurez pas besoin de rechercher d'autres sources pour obtenir les définitions qui vous seront nécessaires.
- » Dans les séquences de commandes, les commandes sur lesquelles vous devez cliquer sont séparées par une flèche spéciale : ainsi, Fichier → Nouveau Fichier signifie qu'il faut cliquer sur Fichier puis sur Nouveau Fichier.

Idées reçues

Il pourra vous sembler difficile de croire que nous nous sommes déjà fait une idée vous concernant, sachant que nous ne nous sommes encore jamais rencontrés ! Les suppositions sont certes un exercice hasardeux, mais nous avons dû en formuler en guise de point de départ pour la rédaction de ce livre.

Nous avons d'abord supposé que vous connaissiez la plateforme que vous comptez utiliser, aussi ce livre ne donne-t-il aucune indication dans ce domaine (toutefois, le [Chapitre 3](#) vous explique comment installer Anaconda, le [Chapitre 4](#) est un aperçu du langage Python, et le [Chapitre 5](#) vous apporte la compréhension nécessaire à l'exécution des manipulations de données essentielles à l'aide de Python). Dans le souci de vous fournir un maximum d'informations sur Python, en ce qui concerne les algorithmes, ce livre n'aborde pas les questions qui sont propres à telle ou telle plateforme. Avant de vous servir de ce livre, il faut vraiment que vous sachiez installer les applications, les utiliser, et de façon générale, travailler sur la plateforme que vous avez choisie.

Ce livre n'est pas destiné à l'apprentissage des mathématiques. Certes, vous y trouverez beaucoup d'exemples de calculs compliqués, mais il s'agit de vous aider à vous servir de Python pour exécuter des tâches courantes au moyen d'algorithmes, et non de vous enseigner la théorie des maths. Néanmoins, nous vous donnons des explications concernant un grand nombre d'algorithmes utilisés dans ce livre, afin que vous puissiez comprendre le principe de leur fonctionnement. Les Chapitres [1](#) et [2](#) vous apportent précisément les bases qui vous permettront de tirer profit de ce livre.

Nous supposons aussi que vous êtes en mesure de naviguer sur l'Internet. Vous trouverez, tout au long de ce livre, de nombreuses références à des contenus en ligne qui enrichiront votre expérience. Cependant, ces sources supplémentaires ne vous

seront utiles que si vous vous donnez la peine de les trouver et de vous en servir.

Icônes utilisées dans ce livre

Au cours de votre lecture, vous trouverez dans la marge des icônes qui vous signaleront quelque chose d'intéressant (ou non, selon le cas).



Ce qui est bien avec ces indications, c'est qu'elles vous permettent d'économiser du temps ou d'exécuter une tâche sans trop de travail supplémentaire. Cette icône signale des techniques qui vous feront gagner du temps et des ressources pour tirer le maximum de Python, lorsque vous devez exécuter des tâches liées aux algorithmes ou à l'analyse de données.



Nous ne voudrions pas avoir l'air de parents en colère ni passer pour des maniaques, mais tout ce qui est signalé par cette icône doit vraiment être évité. Dans le cas contraire, votre application risquerait fort de ne pas fonctionner comme prévu, vous obtiendriez des résultats incorrects malgré des algorithmes apparemment impeccables, ou bien (dans le pire des cas) vous perdriez des données.



Cette icône signale un conseil subtil ou une technique avancée. Il s'agit d'éléments d'information utiles que vous trouverez peut-être parfois rébarbatifs, mais qui peuvent aussi être la solution dont vous avez besoin pour faire fonctionner un programme. Vous pouvez les ignorer si vous le voulez.



Si vous ne deviez retenir qu'une chose d'un chapitre ou d'une section, que ce soit l'information repérée par cette icône. Il s'agira généralement d'un processus essentiel ou d'une connaissance à acquérir pour pouvoir travailler avec Python, ou pour pouvoir réussir des tâches liées à des algorithmes ou à l'analyse de données.

Pour aller plus loin

Votre expérience d'apprentissage de Python et des algorithmes ne doit pas se limiter à ce livre, dont la lecture n'est que le début. Nous vous proposons du contenu en ligne, pour plus de flexibilité et pour que ce livre réponde mieux à vos besoins. Ainsi, si vous nous adressez un courrier électronique, nous pourrions répondre à vos questions et vous préciser de quelle façon les mises à jour de Python et des modules qui lui sont associés affectent le contenu de ce livre. Vous bénéficierez même des avantages suivants :

- » **Une antisèche :** Peut-être vous est-il arrivé, au lycée, d'utiliser une antisèche pour obtenir une meilleure note à un contrôle ? Ici, c'est un peu la même chose. Il s'agit de notes particulières concernant des tâches à effectuer avec Python, avec Anaconda et avec des algorithmes que tout le monde ne connaît pas. Sur le site Internet www.dummies.com, cherchez *Algorithms For Dummies Cheat Sheet*. Cette page recèle des informations utiles pour trouver les algorithmes communément nécessaires à l'exécution de tâches spécifiques.
- » **Des mises à jour :** Il arrive que des changements se produisent. Il se peut qu'un changement imminent nous ait échappé lorsque nous avons consulté notre boule de cristal au cours de la rédaction de ce livre. Dans le passé, vous vous seriez alors retrouvé avec dans les mains un ouvrage périmé, mais aujourd'hui vous pouvez trouver des mises à jour de ce livre sur la page www.dummies.com/go/algorithmsfd.

Outre ces mises à jour, intéressez-vous à la publication des réponses aux questions des lecteurs et aux démonstrations des techniques sur la page <http://blog.johnmuellerbooks.com/>.

» **Des fichiers d'accompagnement :** Qui donc voudrait recopier tout le code contenu dans ce livre et reconstituer à la main toutes ces écritures ? En général, plutôt que de faire de la saisie, les lecteurs préfèrent consacrer leur temps à travailler avec Python, à exécuter des tâches à l'aide des algorithmes et à voir tout ce qu'ils peuvent faire d'intéressant. Heureusement pour vous, les exemples utilisés dans ce livre peuvent être téléchargés. Il vous suffit donc de lire le livre afin d'assimiler les techniques d'utilisation des algorithmes. Vous trouverez les fichiers en question sur la page www.dummies.com/go/algorithmsfd.

Par où commencer ?

Il est temps de vous lancer dans l'apprentissage des algorithmes ! Si les algorithmes sont une chose entièrement nouvelle pour vous, commencez par le [Chapitre 1](#) et poursuivez votre lecture à un rythme vous permettant d'assimiler le plus de connaissances possible. Ne manquez pas de lire ce qui concerne Python, car c'est le langage utilisé dans les exemples.

Si vous êtes novice et si vous bouillez d'impatience de vous lancer dans les algorithmes, vous pouvez passer directement au [Chapitre 3](#), sachant cependant que vous risquez de rencontrer quelques difficultés par la suite. Si vous avez déjà installé Anaconda, vous pouvez parcourir rapidement le [Chapitre 3](#). Avant de mettre en pratique le contenu de ce livre, vous devez installer Python, version 3.4. Les exemples ne fonctionneront pas avec la version 2.x, car celle-ci n'est pas compatible avec certains modules que nous utilisons.

Si vous avez déjà fait connaissance avec Python et si vous avez installé les versions appropriées du langage, passez directement au [Chapitre 6](#) et vous gagnerez du temps. Vous pourrez toujours revenir en arrière autant de fois que nécessaire pour trouver les réponses à vos questions. Cependant, il importe que vous ayez assimilé chaque technique avant d'aborder la suivante. En effet, chacune de ces techniques, chaque exemple de codage et chaque procédure recèlent d'importants enseignements pour vous, et en ignorant toutes ces informations, vous risqueriez de passer à côté de quelque chose d'essentiel.

PARTIE 1

Pour commencer

DANS CETTE PARTIE...

Découvrir comment utiliser des algorithmes pour exécuter des tâches pratiques

Comprendre comment les algorithmes sont construits

Installer et configurer Python pour travailler avec des algorithmes

Utiliser Python pour travailler avec des algorithmes

Commencer à réaliser des manipulations à l'aide de Python

Chapitre 1

Introduction à l'algorithmique

DANS CE CHAPITRE

- » Définir ce qu'est un *algorithme*
 - » Se servir de l'informatique pour produire des solutions grâce aux algorithmes
 - » Déterminer en quoi les questions diffèrent des solutions
 - » Manipuler les données pour trouver une solution
-

Peut-être que comme la majorité des gens, vous êtes un peu perplexe devant la perspective de vous lancer dans la découverte des algorithmes, sachant que la plupart des manuels ne vous disent jamais ce qu'est un algorithme, sans parler de vous expliquer pour quelle raison vous auriez besoin d'en utiliser. La plupart du temps, les auteurs supposent que vous avez déjà des notions d'algorithmique et que vous lisez leurs ouvrages pour approfondir vos connaissances. Curieusement, certains d'entre eux proposent une définition de l'*algorithme* qui porte à confusion et qui, en fait, n'est pas vraiment une définition. Parfois même, ils assimilent l'algorithme à une expression abstraite, numérique ou symbolique.

La première section de ce chapitre vous explique précisément ce que signifie le mot *algorithme* et pourquoi il est dans votre intérêt de savoir utiliser des algorithmes. Loin d'être quelque chose d'obscur, les algorithmes sont utilisés un peu partout, et vous en avez probablement déjà utilisé ou bénéficié à maintes reprises

sans le savoir. En vérité, les algorithmes jouent un rôle toujours plus fondamental dans l'assistance et la régulation de tout ce qui importe le plus dans une société de plus en plus complexe et technologiquement avancée comme la nôtre.

Ce chapitre étudie également la manière dont vous pouvez utiliser l'ordinateur pour trouver des solutions et résoudre des problèmes à l'aide d'algorithmes, il vous indique comment faire la distinction entre les problèmes et les solutions, et il vous explique ce que vous devez faire pour manipuler des données en vue de trouver une solution. L'objectif de ce chapitre est de vous permettre de faire la différence entre les algorithmes et d'autres tâches avec lesquelles ils sont souvent confondus. En un mot, vous allez découvrir une excellente raison de vouloir mieux maîtriser les algorithmes et la méthode pour les appliquer aux données.

Décrire les algorithmes

Cela fait plusieurs millénaires que l'on résout des algorithmes à la main, et cependant, il faut parfois pour cela énormément de temps et de nombreuses opérations de calcul numérique, selon la complexité du problème à résoudre. Les algorithmes servent avant tout à trouver des solutions, et les meilleurs algorithmes sont les plus rapides et les plus faciles. Il existe une différence considérable entre les algorithmes mathématiques mis au point au cours de l'Histoire par des génies comme Euclide, Newton ou Gauss, et les algorithmes actuellement développés dans les universités et dans les laboratoires de recherche. La principale raison de cette différence est que l'on utilise aujourd'hui des ordinateurs. Le recours aux ordinateurs pour résoudre les problèmes en utilisant l'algorithme approprié accélère considérablement la tâche, ce qui explique que le développement de nouveaux algorithmes ait progressé si vite depuis l'apparition de systèmes informatiques performants. Peut-être avez-vous remarqué que de plus en plus, aujourd'hui, les solutions apparaissent rapidement, en partie parce que la puissance de

calcul est à la fois bon marché et en augmentation constante. Compte tenu de leur capacité de résolution de problèmes à l'aide d'algorithmes, les ordinateurs (parfois sous forme d'appareils spéciaux) deviennent omniprésents.

Travailler avec des algorithmes consiste à étudier des inputs (entrées de données), des outputs (sorties) recherchés et un processus (une séquence d'actions) utilisé pour obtenir l'output désiré à partir d'un certain input. Cependant, vous risquez de faire une erreur de terminologie et d'avoir une vision faussée des algorithmes si vous n'avez pas étudié la façon dont ils fonctionnent dans une situation réelle. La troisième section de ce chapitre traite des algorithmes sous un angle concret, c'est-à-dire en reprenant les terminologies utilisées pour les appréhender et en les présentant de manière à traduire le fait qu'une situation réelle est souvent caractérisée par l'imperfection. Savoir décrire un algorithme de façon réaliste est aussi ce qui permet de tempérer les attentes et de refléter ce que cet algorithme peut réellement produire.

Dans ce livre, les algorithmes sont étudiés sous différents angles. Néanmoins, afin de donner un aperçu de la façon dont ils changent et enrichissent la vie des gens, l'accent est mis sur les algorithmes utilisés pour manipuler des données à l'aide d'un ordinateur et exécuter le traitement requis. Dans cette optique, les algorithmes sur lesquels ce livre vous propose de travailler nécessitent la saisie de données sous une forme particulière, si bien que les données doivent parfois être modifiées pour pouvoir correspondre aux spécifications de l'algorithme. La manipulation des données ne consiste pas à changer les données elles-mêmes, mais seulement leur présentation et leur forme, de manière à ce que l'algorithme puisse faire apparaître de nouvelles tendances qui, auparavant, n'étaient pas visibles (bien que déjà présentes dans les données).

Les sources d'information sur les algorithmes les présentent souvent de façon confuse : ils apparaissent trop complexes, quand ils ne sont pas carrément incorrects. Concernant les algorithmes et les autres notions avec lesquelles ils sont souvent confondus (à

tort), nous nous en tiendrons dans ce livre aux définitions suivantes, même s'il en existe d'autres :

» **Équation** : Séquence de nombres et de symboles qui, ensemble, forment une égalité avec une valeur spécifique. Une équation comporte toujours un signe égal, si bien que l'on sait que le terme constitué de nombres et de symboles représente la valeur spécifique écrite de l'autre côté de ce signe. Une équation comporte généralement des éléments variables représentés sous forme de symboles, mais elle ne comporte pas nécessairement des variables.

» **Formule** : Combinaison de nombres et de symboles utilisés pour représenter une information ou une idée. Normalement, les formules représentent des concepts mathématiques ou logiques, comme la définition du plus grand commun diviseur (PGCD) de deux entiers (expliquée dans la vidéo suivante :

<https://www.khanacademy.org/math/in-sixth-grade-math/playing-numbers/highest-common-factor/v/greatest-common-divisor>). De façon générale, elles montrent la relation entre deux ou plusieurs variables. La plupart du temps, les formules sont considérées comme un type particulier d'équation.



» **Algorithme** : succession d'étapes destinée à résoudre un problème. Il s'agit d'une séquence représentant une méthode unique de résolution d'un problème par la production d'une solution. Un algorithme ne représente pas nécessairement des concepts mathématiques ou logiques, même si c'est souvent le cas dans ce livre, sachant que les algorithmes sont communément utilisés de cette manière. Certaines formules sont aussi des algorithmes, comme par exemple la formule quadratique. Un processus est un algorithme lorsqu'il présente les propriétés suivantes :

- **Il est fini** : L'algorithme doit résoudre le problème. Ce livre traite de problèmes dont la solution est connue, de telle sorte que vous pouvez vérifier

qu'un algorithme permet de résoudre correctement chaque problème.

- **Il est bien défini** : Les étapes doivent se succéder de façon précise et elles doivent être compréhensibles. Sachant que l'ordinateur est utilisé, celui-ci doit pouvoir interpréter chaque étape et produire un algorithme utilisable.
- **Il est efficace** : L'algorithme doit pouvoir traiter toutes les occurrences du problème pour la résolution duquel il a été défini. Un algorithme doit toujours résoudre le problème qu'il est censé résoudre. Certes, il importe d'anticiper certains échecs, mais les échecs sont rares et ne se produisent que dans des situations qui sont acceptables dans le contexte de l'utilisation prévue de l'algorithme.

Compte tenu de ces définitions, les sections qui suivent permettent de clarifier la nature précise des algorithmes. L'objectif n'est pas d'obtenir une définition précise des algorithmes, mais plutôt de vous aider à percevoir le rôle que jouent les algorithmes dans le grand ordre des choses, afin que vous puissiez vous faire votre propre idée de ce que sont les algorithmes et comprendre pourquoi ils sont si importants.

Définir les utilisations possibles des algorithmes

Un algorithme se présente toujours sous la forme d'une série d'étapes, par lesquelles il ne passera pas nécessairement pour résoudre une formule mathématique. Le champ des algorithmes est extraordinairement vaste. Les algorithmes sont utilisés pour résoudre des problèmes en science, en médecine, en finance, dans le domaine de la production industrielle et de l'approvisionnement, et dans la communication. Les algorithmes

nous sont utiles à plus d'un titre dans notre quotidien. Toute série d'actions en vue d'un résultat qui est finie, précisément définie et efficace, peut être considérée comme un algorithme. Ainsi, par exemple, même une chose aussi simple et aussi banale que préparer des toasts peut être représentée sous forme d'algorithme. Une telle procédure sert souvent d'illustration dans les cours d'informatique, comme cela apparaît sur la page Internet <http://brianaspinall.com/now-thats-how-you-make-toast-using-computer-algorithms/>.



Malheureusement, sur le site Internet en question, l'algorithme est faux. L'instructeur ne retire jamais le pain de mie de son emballage et ne met jamais le grille-pain sous tension. Par conséquent, le résultat obtenu, c'est du pain de mie nature gâché, qui aura été fourré avec son emballage en plastique dans un grille-pain non fonctionnel (pour plus de détails, voir la page <http://blog.johnmuellerbooks.com/2013/03/04/procedures-in-technicalwriting/>). Ce n'est pas l'idée qui est en cause, cependant il faudrait procéder à quelques corrections, légères mais essentielles, afin que l'algorithme soit fini et efficace.

Une des applications les plus courantes de l'algorithmique est la résolution des formules. Déterminer le PGCD de deux entiers, par exemple, est une tâche que vous pouvez exécuter manuellement en recensant les facteurs des deux entiers puis en sélectionnant le plus grand des facteurs qui leur sont communs. Ainsi, le PGCD de 20 et 25 est 5, car parmi les diviseurs de 20 et de 25, c'est 5 qui est le plus grand. Cependant, la détermination manuelle des PGCD (qui est bel et bien une sorte d'algorithme) demande du temps et peut être source d'erreurs. C'est pourquoi le mathématicien grec Euclide (<https://fr.wikipedia.org/wiki/Euclide>) a inventé un algorithme destiné à exécuter cette tâche. Vous pouvez voir une démonstration de la méthode euclidienne sur la page <https://www.khanacademy.org/computing/computerscience/crypto-euclidean-algorithm>.

À une formule unique, constituée de symboles et de nombres servant à exprimer une information ou une idée, peuvent

correspondre des solutions multiples, chacune constituant un algorithme. Concernant le PGCD, un autre algorithme couramment utilisé est celui inventé par Derrick Henry Lehmer (pour plus de détails, voir <https://www.imsc.res.in/~kapil/crypto/notes/node11.html> et https://en.wikipedia.org/wiki/Lehmer%27s_GCD_algorithm).

Sachant que toute formule peut être résolue de plusieurs manières, on passe souvent beaucoup de temps à comparer les algorithmes en vue de déterminer lequel est le plus adapté dans une situation donnée (pour une comparaison entre Euclide et Lehmer, voir <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.693&rep=rep1&type=pdf>).



L'évolution de plus en plus rapide de notre société et de ses technologies entraîne de nouveaux besoins d'algorithmes. De nos jours, des avancées scientifiques comme le séquençage du génome humain n'ont été possibles que parce que les scientifiques ont pu mettre au point des algorithmes assez performants pour exécuter ce type de tâche. Déterminer quel algorithme est le plus approprié dans une situation donnée ou dans une situation type est une question sérieuse et fait l'objet de débats chez les informaticiens.

Dans le domaine de l'informatique, le même algorithme peut être présenté de multiples façons. L'algorithme d'Euclide, par exemple, peut être présenté sous forme récursive ou bien sous forme itérative, comme cela est expliqué sur la page <http://cs.stackexchange.com/questions/1447/what-is-most-efficient-for-gcd>. En résumé, les algorithmes représentent une méthode de résolution des formules, mais ce serait une erreur que de croire qu'un seul algorithme est acceptable pour une formule donnée ou qu'il n'existe qu'une façon acceptable de représenter un algorithme. L'utilisation d'algorithmes pour résoudre des problèmes de diverses sortes remonte loin dans l'Histoire : ce n'est certainement pas une nouveauté.

Même en limitant le champ de l'étude à des domaines techniques comme l'informatique, la science des données ou l'intelligence artificielle, on peut trouver toutes sortes d'algorithmes : de quoi remplir plusieurs ouvrages. L'ouvrage de Donald E. Knuth *The*

Art of Computer Programming (Addison-Wesley), par exemple, totalise 3 168 pages réparties sur quatre volumes (voir <http://www.amazon.com/exec/obidos/ASIN/0321751043/datacserve> 20/) et ne fait pas pour autant le tour du sujet (l'auteur avait prévu de publier encore d'autres volumes). Voici néanmoins quelques applications qu'il peut être intéressant pour vous d'envisager :

- » **La recherche d'informations** : Trouver des informations et vérifier leur pertinence constituent une tâche essentielle. Sans cela, un certain nombre de tâches ne pourraient pas être exécutées en ligne, comme par exemple trouver le site Web qui vend le modèle de cafetière convenant le mieux pour votre bureau.
- » **Le tri de l'information** : Il est important de déterminer l'ordre dans lequel l'information doit être présentée. En effet, aujourd'hui la plupart des gens sont saturés d'informations, et le tri de l'information permet de réduire l'excès de données. Enfant, vous aviez sans doute appris qu'il est plus facile de trouver le jouet qui vous intéresse quand tous les jouets sont à leur place, plutôt qu'éparpillés un peu partout et n'importe comment. Imaginez que plus de mille cafetières différentes soient en vente sur Amazon et qu'il ne soit pas possible de les trier en fonction du prix ni des avis. Par ailleurs, les algorithmes compliqués ne peuvent fonctionner de façon fiable que si les données sont entrées dans le bon ordre. Le tri des données est donc un préalable important à la résolution des problèmes.
- » **La transformation des données** : Pour que les données soient comprises et utilisées de façon satisfaisante, il est essentiel de les convertir. Ainsi, par exemple, même si vous avez une bonne connaissance du système impérial des poids et mesures et si vous avez tendance à exprimer les poids en livres, vous ne pouvez pas ignorer que toutes vos sources utilisent le système métrique. De même, la transformation de Fourier rapide (*Fast Fourier Transform* ou FFT) convertit les signaux pour les faire passer du domaine

temporel au domaine fréquentiel, afin de permettre, par exemple, le fonctionnement de votre routeur Wifi.

- » **L'ordonnancement** : Pour permettre à toutes les parties concernées d'utiliser les ressources, les algorithmes jouent aussi un grand rôle. Les feux de signalisation aux intersections, par exemple, ne sont plus de simples compteurs qui déclenchent les changements de feux au bout d'un certain nombre de secondes. Les installations actuelles prennent en compte divers paramètres comme le moment de la journée, la météo et le flux de la circulation. Cependant, il existe plusieurs sortes de programmation. Songez, par exemple, à la manière dont votre ordinateur exécute plusieurs tâches en même temps. Sans un algorithme d'ordonnancement, le système d'exploitation risquerait de mobiliser toutes les ressources disponibles et votre application serait inutilisable.
- » **L'analyse graphique** : La détermination du plus court chemin entre deux points trouve toutes sortes d'applications. Votre GPS, par exemple, ne pourrait pas fonctionner sans un algorithme de ce type qui détermine le trajet le plus court entre un point A et un point B.
- » **La cryptographie** : Sachant que les sources de données font sans cesse l'objet d'attaques par des hackers, la sécurité des données est un combat permanent. C'est grâce à des algorithmes que les ordinateurs peuvent analyser les données, leur donner une autre forme qui ne soit pas lisible, et leur rendre par la suite leur format initial.
- » **La génération de nombres pseudo-aléatoires** : Pouvez-vous imaginer un jeu dans lequel vous commenceriez à jouer toujours à la même place, effectueriez toujours le même parcours et franchiriez toujours les mêmes étapes, de la même manière, à chaque partie ? Sans la possibilité de générer des nombres apparemment aléatoires, de nombreux traitements informatiques seraient impossibles.



Cette liste ne constitue qu'un très court aperçu. Les algorithmes servent à effectuer un nombre considérable de tâches très variées, de manières différentes, et de nouveaux algorithmes sont développés constamment pour résoudre aussi bien de nouveaux problèmes que des problèmes déjà existants. Le plus important, quand on utilise des algorithmes, est de prendre en compte le fait qu'à une entrée particulière doive correspondre un résultat spécifique. Secondairement, il s'agit de déterminer les ressources nécessaires à l'algorithme et le temps nécessaire pour que la tâche soit exécutée. Selon le type de problème à résoudre et le type d'algorithme utilisé, il se peut que vous ayez à prendre également en compte des considérations de pertinence et de cohérence.

Trouver des algorithmes partout

Si l'exemple des toasts est mentionné dans la section précédente, c'est pour une raison bien particulière. Il se trouve que la préparation des toasts est un des algorithmes les plus connus entre tous. À l'école, les enfants maîtrisent souvent l'équivalent de cet algorithme bien avant même de savoir résoudre le problème de mathématiques le plus élémentaire. Il n'est pas difficile d'imaginer un grand nombre de variantes de l'algorithme du toast, avec le résultat précis de chacune. Certes, les résultats sont susceptibles de varier selon la personne et selon sa créativité. Pour faire court, il existe une variété considérable d'algorithmes et on les retrouve souvent là où on ne les attendait pas.

Toute tâche exécutée à l'aide d'un ordinateur met en jeu des algorithmes. Certains algorithmes sont intégrés au matériel informatique (au niveau des microprocesseurs). Même la simple mise en route de l'ordinateur fait appel à un algorithme. On trouve aussi des algorithmes dans les systèmes d'exploitation, dans les applications informatiques et dans tout logiciel. Même les utilisateurs appliquent des algorithmes. Des scripts leur permettent d'exécuter des tâches d'une manière spécifique, selon des étapes qui peuvent être matérialisées par des instructions écrites ou par une procédure écrite.

Souvent, des algorithmes interviennent dans notre quotidien. Comment se passe votre journée ? Pour l'essentiel, il y a de bonnes chances pour que vous exécutiez tous les jours les mêmes tâches dans le même ordre. On pourrait presque dire que votre journée est un algorithme destiné à résoudre le problème qui consiste à mener une vie satisfaisante tout en dépensant le moins d'énergie possible. Finalement, c'est bien le principe de la routine : c'est ce qui nous rend efficaces.

Les procédures d'urgence reposent souvent sur des algorithmes. Dans l'avion, vous devez commencer par sortir la plaquette qui se trouve dans la poche devant vous. Elle comporte une série d'instructions illustrées pour ouvrir l'issue de secours et dérouler la glissière. Dans certains cas, il n'y a même pas de texte, toute la procédure requise pour exécuter la tâche et résoudre le problème, qui consiste à sortir précipitamment de l'avion, est expliquée par les dessins. Tout au long de ce livre, vous retrouverez les trois mêmes éléments, quel que soit l'algorithme :

1. **Description du problème.**
2. **Élaboration d'une série d'étapes (bien définies) pour résoudre le problème.**
3. **Exécution des étapes pour obtenir le résultat désiré (processus fini et efficace).**

Utiliser l'informatique pour résoudre des problèmes

Le mot *informatique* a une connotation très technique et peut rebuter certaines personnes, mais aujourd'hui nous sommes tous plongés dans l'informatique jusqu'au cou (ou plus encore). La plupart du temps, vous avez sous la main au moins un ordinateur, c'est-à-dire votre smartphone. Si jamais vous portez un pacemaker, par exemple, sachez que cet appareil aussi est informatisé. Votre téléviseur « intelligent » est équipé d'au moins

un dispositif informatique, et il en est de même de votre four programmable. Une automobile peut comporter pas moins de 30 calculateurs électroniques sous forme de microprocesseurs intégrés régulant la consommation de carburant, la combustion, la transmission (automatique), la direction et la stabilité du véhicule (d'après un article du *New York Times* à l'adresse <http://www.nytimes.com/2010/02/05/technology/05electronics.htm> et tout cela peut représenter davantage de lignes de code que l'équipement électronique d'un avion de combat. Les véhicules automatisés qui vont bientôt apparaître sur le marché de l'automobile utiliseront encore plus de microprocesseurs intégrés et des algorithmes encore plus élaborés. La finalité d'un ordinateur est de résoudre des problèmes rapidement et avec moins d'efforts qu'une résolution manuelle. Il n'est donc pas surprenant que ce livre ait nécessité davantage encore d'ordinateurs pour vous permettre de mieux comprendre l'algorithmique.

Il existe toutes sortes d'ordinateurs et de calculateurs électroniques. Celui de votre montre est minuscule, tandis que votre ordinateur de bureau occupe une certaine place. Les supercalculateurs sont immenses et sont constitués d'un grand nombre de petits ordinateurs qui sont programmés pour travailler ensemble afin de résoudre des problèmes complexes, par exemple déterminer le temps qu'il fera demain. Les algorithmes les plus élaborés font appel à des fonctionnalités informatiques spécifiques afin d'aboutir à des solutions aux problèmes qu'ils sont conçus pour résoudre. Oui, vous pourriez utiliser moins de ressources pour exécuter une tâche donnée, mais la réponse se ferait attendre bien plus longtemps, ou bien elle ne serait pas assez précise pour constituer une solution exploitable. Dans certains cas, le temps d'attente serait si long que la réponse, une fois obtenue, n'aurait plus d'importance. Compte tenu des impératifs de rapidité et de fiabilité, les sections qui suivent sont consacrées à certains aspects particuliers de l'informatique dont les algorithmes peuvent dépendre.

Exploiter les processeurs généralistes et les processeurs graphiques

Les processeurs généralistes, ou unités centrales de traitement (UCT, en anglais CPU), ont été conçus pour résoudre des problèmes à l'aide d'algorithmes. Cependant, de par leur polyvalence, ils peuvent servir à un grand nombre d'autres tâches, comme transférer des données ou assurer des interactions avec des systèmes extérieurs. Ils peuvent donc exécuter les étapes d'un algorithme, mais pas toujours très rapidement. Il était possible d'adjoindre aux premiers processeurs un coprocesseur mathématique (une puce spécialisée dans le calcul) afin de gagner en rapidité (pour plus de détails, voir <http://www.computerhope.com/jargon/m/mathcopr.htm>), mais aujourd'hui les processeurs généralistes sont dotés d'un coprocesseur mathématique intégré. Un ordinateur équipé d'un processeur Intel i7, par exemple, comporte en réalité plusieurs processeurs sous forme d'un module unique.



Curieusement, Intel commercialise toujours des processeurs complémentaires spécialisés, comme le processeur Xeon Phi qui fonctionne avec les puces Xeon (pour plus de détails, voir <http://www.intel.fr/content/www/fr/fr/products/processors/xeon-phi/xeon-phi-processors.html> et https://en.wiki2.org/wiki/Intel_Xeon_Phi). La puce Xeon Phi s'utilise en combinaison avec une puce Xeon pour effectuer des tâches intensives en calculs comme l'apprentissage machine (à propos de l'utilisation des algorithmes dans l'apprentissage machine pour déterminer le processus d'exécution de tâches diverses et pour prédire l'inconnu et organiser l'information).

Vous vous demandez peut-être pourquoi le titre de cette section mentionne les processeurs graphiques (unités de traitement graphique, GPU en anglais). C'est que les GPU utilisent des données, les manipulent d'une certaine façon, puis affichent une

jolie image sur l'écran. Tout matériel informatique peut avoir plus d'une finalité. Il se trouve que les processeurs graphiques sont particulièrement adaptés à la transformation des données, une tâche essentielle dans bien des cas pour résoudre les algorithmes. Un processeur graphique est un processeur spécialisé, mais qui est conçu pour exécuter les algorithmes plus rapidement. Ne soyez pas surpris de découvrir que les inventeurs d'algorithmes consacrent beaucoup de temps à penser différemment et à exercer leur créativité, et trouvent des méthodes de résolution de problèmes en adoptant des approches non traditionnelles.

Ce qu'il faut savoir, c'est que les processeurs généralistes et les processeurs graphiques sont les puces les plus couramment utilisées pour l'exécution de tâches fondées sur des algorithmes. Les premiers s'acquittent très bien des tâches polyvalentes, tandis que les seconds servent surtout à faciliter les tâches intensives en calculs mathématiques, surtout lorsqu'il s'agit d'effectuer des transformations de données. L'utilisation de processeurs multiples rend possible le traitement en parallèle (c'est-à-dire l'exécution de plusieurs étapes algorithmiques à la fois). Plus il y a de puces, plus on peut avoir d'unités de traitement, ce qui permet d'augmenter la vitesse de traitement, mais un certain nombre de facteurs limitent ce gain de rapidité. L'utilisation de deux processeurs i7 ne permet pas d'obtenir une vitesse deux fois plus élevée que celle d'un seul processeur i7.

Travailler avec des processeurs spécialisés

Le coprocesseur mathématique et le processeur graphique sont deux exemples de processeurs spécialisés d'utilisation courante. Ils ne servent pas à effectuer des tâches comme faire démarrer le système. Cependant, la résolution de problèmes à l'aide d'algorithmes implique souvent le recours à des processeurs spécialisés. Ce livre n'est pas censé traiter du matériel informatique, mais il n'est pas inutile d'y consacrer un peu de

temps et de découvrir toutes sortes de puces intéressantes, comme les nouveaux neurones artificiels sur lesquels travaille actuellement IBM (voir

<http://www.computerworld.com/article/3103294/computer-processors/ibm-creates-artificial-neurons-from-phase-changememory-for-cognitive-computing.html>). Imaginons un

processus algorithmique utilisant une mémoire qui simule le cerveau humain. On obtiendrait ainsi un contexte intéressant pour l'exécution de tâches qui, autrement, pourraient ne pas être envisageables aujourd'hui.

Les réseaux neuronaux, une technologie utilisée pour simuler la pensée humaine et rendre possibles des techniques d'apprentissage poussé dans le cadre de projets d'apprentissage machine, bénéficient maintenant de l'utilisation de processeurs spécialisés comme le Tesla P100 de NVidia (pour plus de détails, voir <https://www.technologyreview.com/s/601195/a-2-billionchip-to-accelerate-artificial-intelligence/>).

Non seulement ces processeurs exécutent des traitements algorithmiques à des vitesses extrêmement rapides, mais ils apprennent à mesure qu'ils exécutent les tâches, si bien qu'ils deviennent plus rapides à chaque itération. Un jour, les ordinateurs capables d'apprendre feront fonctionner des robots qui se déplaceront eux-mêmes (d'une certaine façon), comme ceux que l'on voit dans le film *I, Robot*. Il existe aussi des processeurs spéciaux qui exécutent des tâches comme la reconnaissance visuelle.



Les processeurs spécialisés, qui peuvent fonctionner de diverses manières, appliqueront un jour toutes sortes d'algorithmes avec des conséquences dans le monde réel. Certaines applications sont déjà observables sous une forme relativement simple. Imaginons, par exemple, les tâches que devrait exécuter un robot conçu pour fabriquer des pizzas, et les variables qu'il devrait prendre en compte en temps réel. Ce genre de robot existe déjà (ce n'est qu'un exemple parmi d'autres de robots industriels utilisés pour produire des biens matériels à l'aide d'algorithmes), et bien évidemment, ce sont des algorithmes qui décrivent ce qu'il faut faire, et ce sont des processeurs spécialisés qui assurent

l'exécution rapide des tâches.



Il sera peut-être même possible un jour de se servir de l'esprit humain comme d'un processeur et de produire l'information à l'aide d'une interface particulière. Certaines entreprises procèdent aujourd'hui à des expérimentations consistant à implanter des processeurs directement dans le cerveau humain afin de renforcer sa capacité de traitement de l'information. On peut imaginer un système dans lequel des humains pourraient exécuter des algorithmes à la vitesse des ordinateurs, mais avec le potentiel de créativité propre à l'être humain.

Tirer parti des réseaux

À moins de disposer de fonds illimités, il n'est pas toujours possible d'utiliser des algorithmes de façon rentable, même avec des processeurs spécialisés. C'est pourquoi il est intéressant de pouvoir exploiter les ordinateurs en réseau. Grâce à un logiciel spécial, un ordinateur « maître », ou ordinateur pilote, peut utiliser les processeurs de tous les ordinateurs esclaves. Il utilise pour cela un *agent* (sorte d'application en arrière-plan qui assure la disponibilité du processeur). Cette approche permet de résoudre des problèmes extrêmement compliqués, en confiant une partie des tâches à un certain nombre d'ordinateurs. Chaque ordinateur du réseau accomplit sa part du travail et renvoie les résultats à l'ordinateur maître, lequel rassemble les éléments pour former une réponse unifiée, une technique appelée l'informatique en *grappe*.

Cela peut ressembler à de la science-fiction, mais ces techniques informatiques font déjà l'objet de toutes sortes d'applications.

L'*informatique distribuée* est une autre version courante de l'informatique en grappe (mais avec une organisation moins stricte). Vous trouverez une liste de projets d'informatique distribuée sur la page <http://www.distributedcomputing.info/projects.html>. Cette liste comporte des réalisations majeures comme Search for Extraterrestrial Intelligence (SETI). Vous pouvez aussi ajouter de

la puissance de calcul à votre ordinateur pour travailler sur un traitement du cancer. Les possibilités de projets sont considérables.

Les réseaux vous permettent aussi d'accéder à la puissance de traitement d'autres utilisateurs de façon moins codifiée. Ainsi, Amazon Web Services (AWS) et d'autres sociétés mettent leurs ordinateurs à votre disposition. Grâce à une connexion en réseau, tout se passe comme si ces ordinateurs distants faisaient partie de votre propre réseau informatique. L'idée est que le réseautage peut servir, sous différentes formes, à créer des connexions entre des ordinateurs pour exécuter divers algorithmes qu'il serait trop compliqué d'exécuter à l'aide de votre système seul.

Exploiter les données disponibles

L'exécution d'un algorithme ne dépend pas que de la puissance de traitement, de la pensée créative et d'éléments physiques. Pour trouver une solution à la plupart des problèmes, il faut aussi des données sur lesquelles fonder une conclusion. Ainsi, dans l'exemple de la préparation des toasts, il faut d'abord connaître la disponibilité du pain, il faut un grille-pain, il faut une source électrique pour alimenter l'appareil, *etc.* Ce n'est qu'ensuite qu'il est possible de résoudre le problème de la préparation des toasts. Les données sont importantes, car il n'est pas possible de terminer l'exécution de l'algorithme s'il manque ne serait-ce qu'un élément de la solution. Bien sûr, l'entrée de données supplémentaires peut être nécessaire. Ainsi, par exemple, la personne qui désire un toast n'aime pas forcément le pain de seigle. Si c'est le cas et si vous n'avez sous la main que du pain de seigle, alors la présence du pain ne suffira pas à garantir un résultat satisfaisant.

Les données peuvent provenir de toutes sortes de sources, sous toutes sortes de formes. Vous pouvez transférer des données provenant d'un système de surveillance en temps réel, accéder à une source publique de données, exploiter des données privées contenues dans une base de données, recueillir des données sur

différents sites Web, et les autres possibilités sont encore trop nombreuses pour pouvoir être mentionnées ici. Les données peuvent être statiques (si elles ne changent pas) ou dynamiques (si elles changent constamment). Elles peuvent être complètes ou incomplètes. Elles n'ont pas nécessairement le format désiré (s'il s'agit, par exemple, de poids exprimés en livres et en tonnes américaines alors que vous avez besoin de données exprimées en unités du système métrique). Les données peuvent être présentées sous forme de tableau alors qu'il vous les faudrait sous une autre forme. Elles peuvent se présenter de façon non structurée (par exemple dans une base de données NoSQL ou sous forme d'une série de fichiers de données) alors que vous auriez besoin du format normal d'une base de données relationnelle. En un mot, pour pouvoir résoudre des problèmes à partir des données qui seront exploitées par votre algorithme, vous avez besoin de connaître un certain nombre de choses.



Les données pouvant se présenter sous tant de formats différents, et parce que vous pouvez avoir besoin de les exploiter de tant de façons différentes, ce livre leur accorde beaucoup d'attention. En lisant le [Chapitre 6](#), vous découvrirez le rôle que joue la structure des données. Au [Chapitre 7](#), vous étudierez la recherche des données, afin d'être en mesure de trouver ce dont vous aurez besoin. Les Chapitres [12](#) à [14](#) vous expliquent comment travailler avec des données en grand nombre (des *big data*). Cependant, vous trouverez des informations relatives aux données dans tous les chapitres de ce livre, sachant qu'en l'absence de données, un algorithme ne permettrait de résoudre aucun problème.

Distinguer les problèmes et les solutions

Ce livre aborde deux aspects essentiels de l'algorithmique. Il y a, d'une part, les problèmes à résoudre. Cela peut prendre la forme d'une description du résultat désiré d'un algorithme, ou de la description d'une difficulté à surmonter pour obtenir le résultat

désiré. Les solutions sont les méthodes ou les étapes à utiliser pour résoudre les problèmes. Une solution peut correspondre à une ou plusieurs étapes de l'algorithme. En effet, le résultat d'un algorithme, qui est la solution, est la réponse à la dernière étape. Les sections qui suivent visent à mieux comprendre certains aspects importants de ces problèmes et de ces solutions.

Être correct et efficace

On utilise un algorithme pour obtenir une réponse acceptable. La raison pour laquelle on recherche une réponse qui soit acceptable est que certains algorithmes produisent plus d'une réponse à l'entrée de données trop vagues. La vie est ainsi faite qu'il est parfois impossible d'obtenir des réponses précises. Naturellement, obtenir une réponse précise est toujours l'objectif, mais il arrive souvent que l'on doive se contenter d'une réponse acceptable.

Obtenir la réponse la plus précise possible prend parfois trop de temps. Quand une réponse précise est obtenue trop tard pour pouvoir être exploitée, l'information devient inutile et vous avez perdu votre temps. Le choix entre deux algorithmes pour la résolution du même problème n'est parfois rien d'autre qu'un choix entre la rapidité du traitement et la précision du résultat. Un algorithme rapide ne produira pas toujours une réponse précise, mais cette réponse fera peut-être l'affaire si le résultat est exploitable.

Les réponses fausses peuvent poser un problème. Obtenir rapidement des réponses fausses en grand nombre ne vaut pas mieux qu'obtenir des réponses précises au bout d'un temps plus long. L'objet de ce livre est aussi de vous aider à trouver le juste milieu entre trop rapide et trop lent, et entre imprécis et trop précis. Même si votre professeur de mathématiques soulignait la nécessité de fournir la réponse correcte de la manière exposée dans le livre que vous utilisiez alors, dans le monde réel les mathématiques consistent souvent à évaluer les options et à

prendre des décisions sur la base de compromis, sous des formes que vous n'auriez pas cru possibles.

Où l'on constate que rien n'est gratuit

Vous avez peut-être entendu parler de cette idée reçue, selon laquelle tout doit pouvoir être obtenu des ordinateurs sans devoir faire beaucoup d'efforts pour trouver la solution. Malheureusement, tout problème n'a pas une solution absolue et les meilleures réponses sont souvent onéreuses. En travaillant sur les algorithmes, on ne tarde pas à se rendre compte de la nécessité de disposer de ressources supplémentaires lorsque des réponses précises doivent être obtenues rapidement. Par ailleurs, la dimension et la complexité des sources de données que vous utilisez influent considérablement sur l'obtention de la solution. Plus elles sont vastes et complexes, plus il est nécessaire d'y consacrer davantage de ressources.

Adapter la stratégie au problème

La Cinquième partie de ce livre porte sur les stratégies que vous pouvez employer pour que l'utilisation d'algorithmes vous coûte moins cher. Les meilleurs mathématiciens ont recours à des trucs pour obtenir davantage de résultats avec moins de traitement informatique. Pour résoudre un problème, vous pouvez soit concevoir l'algorithme idéal, soit utiliser une série d'algorithmes plus simples et plusieurs processeurs. En général, cette dernière méthode est meilleure et plus rapide, même si l'approche semble contre-intuitive.

Décrire des algorithmes en lingua franca

Les algorithmes constituent une base pour la communication entre les gens, même entre des personnes dont les points de vue diffèrent et qui ne parlent pas la même langue. Que l'on parle l'anglais, le français, l'espagnol, le chinois, l'allemand ou n'importe quelle autre langue, le théorème de Bayes, par exemple.

Dans tous les cas, l'algorithme a le même aspect et fonctionne de la même manière, les données étant les mêmes. Les algorithmes permettent de passer au travers de toutes sortes de divergences pouvant séparer les gens, car ils expriment des idées sous une forme que tout le monde peut justifier. D'un chapitre à l'autre de ce livre, vous allez découvrir la beauté et la magie que les algorithmes peuvent produire en traduisant même des pensées subtiles.



Abstraction faite de la notation mathématique universelle, les algorithmes bénéficient des langages de programmation en tant que forme d'explication et de communication des formules à résoudre. Il existe toutes sortes d'algorithmes écrits dans des langages comme le C, le C++, Java, le Fortran, Python (comme dans ce livre), *etc.* Certains auteurs utilisent un pseudo-code pour présenter un algorithme autrement que dans un langage de programmation que le lecteur risquerait de ne pas connaître. Le *pseudo-code* est un moyen de décrire les opérations de l'ordinateur à l'aide d'un vocabulaire courant, en français par exemple.

Face à un problème difficile

Une remarque importante à propos du travail sur les algorithmes est que vous pouvez vous en servir pour résoudre des problèmes, quelle qu'en soit la complexité. Un algorithme ne pense pas, il n'a pas d'émotions et il ne se soucie pas de la façon dont vous l'utilisez (ni même, de la façon dont vous le maltraitez). Vous pouvez utiliser des algorithmes de quelque manière que ce soit pour résoudre un problème. Ainsi, par exemple, une même série d'algorithmes utilisée pour la reconnaissance faciale en guise

d'alternative aux mots de passe à saisir sur les ordinateurs (pour des raisons de sécurité) peut permettre d'identifier des terroristes qui rôdent dans un aéroport ou de reconnaître un enfant perdu qui erre dans la rue. Un même algorithme peut se prêter à différentes utilisations, selon les intérêts de l'utilisateur. S'il est conseillé de lire ce livre attentivement, c'est notamment dans l'objectif d'être en mesure de résoudre des problèmes difficiles pour lesquels un algorithme simple peut parfois suffire.

Structurer les données pour obtenir une solution

Les humains envisagent les données de façon non spécifique et appliquent diverses règles aux mêmes données afin de les appréhender d'une manière dont les ordinateurs ne sont jamais capables. Un ordinateur conçoit les données de façon structurée, simple, inflexible, et certainement pas créative. Quand les humains préparent les données que l'ordinateur devra traiter, il arrive souvent que l'interaction entre ces données et les algorithmes prenne un tour inattendu et produise un résultat indésirable. Le problème est que l'être humain ne se rend pas toujours bien compte de la vision limitée des données qui est celle de l'ordinateur. Les sections qui suivent décrivent deux aspects des données dont vous trouverez des illustrations dans un certain nombre des chapitres suivants.

Comprendre le point de vue de l'ordinateur

L'ordinateur a des données une vision simple, mais c'est aussi quelque chose que les gens ne comprennent généralement pas. Déjà, pour l'ordinateur, tout est nombre, car l'ordinateur n'est pas conçu pour traiter d'autres types de données. Les utilisateurs qui voient s'afficher des caractères sur leur écran supposent que

l'ordinateur traite ce type de données, alors que l'ordinateur ne comprend ni les données ni leurs implications. Pour un ordinateur, la lettre *A* n'est pas autre chose que le nombre 65. En fait, ce n'est même pas vraiment le nombre 65. Pour l'ordinateur, c'est une série d'impulsions électriques qui équivaut à la valeur en chiffres binaires 0100 0001.

De même, l'ordinateur ignore la notion de caractère majuscule ou minuscule. Pour nous, la lettre minuscule *a* est simplement une autre forme de la lettre majuscule *A*, mais pour l'ordinateur ce sont deux caractères différents : la lettre minuscule *a* est le nombre 97, c'est-à-dire, en chiffres binaires, 0110 0001.

Si ce genre de comparaison simple entre des caractères d'imprimerie peut poser de tels problèmes entre les humains et les ordinateurs, il n'est pas difficile d'imaginer ce qui peut se produire avec d'autres types de données. Ainsi, par exemple, un ordinateur ne peut pas entendre ni apprécier de la musique. Or, c'est bien de la musique que diffusent les haut-parleurs de votre ordinateur. Il en est de même pour les images. Pour l'ordinateur, une image n'est pas un beau paysage de campagne, mais simplement une série de 0 et de 1.



Quand on utilise des algorithmes, il est important d'envisager les données du point de vue de l'ordinateur. Pour l'ordinateur, il n'existe que des 0 et des 1, et rien d'autre. C'est donc de cette manière que vous devez considérer les données à traiter par un algorithme. Vous vous apercevrez peut-être qu'en adoptant le point de vue de l'ordinateur, les solutions deviennent plus faciles à trouver. En avançant dans votre lecture, vous en saurez davantage sur cette apparente bizarrerie.

Organiser les données change beaucoup de choses

L'ordinateur a aussi une conception stricte de la forme et de la structure des données. En commençant à travailler sur les

algorithmes, vous constaterez qu'une grande partie du travail, lorsqu'il s'agit de trouver une solution à un problème, consiste à mettre l'information sous une forme exploitable par l'ordinateur. Un être humain peut percevoir une tendance dans des données qui ne sont pas bien organisées, mais un ordinateur ne pourra déceler la même tendance que si les données sont organisées de façon très précise. L'avantage de cette précision est que l'ordinateur est souvent capable de rendre visibles de nouvelles tendances. C'est même une des principales raisons pour lesquelles on utilise des algorithmes en informatique : pour déceler de nouvelles tendances, puis les exploiter pour exécuter d'autres tâches. Ainsi, par exemple, un ordinateur peut déterminer le profil de dépenses d'un consommateur, et cette information peut être exploitée pour accroître les ventes de façon automatique.

Chapitre 2

Étude de la conception des algorithmes

DANS CE CHAPITRE

- » Réfléchir à la manière de résoudre un problème
 - » Adopter une approche de type « diviser pour régner » pour résoudre les problèmes
 - » Comprendre l'approche gloutonne de la résolution des problèmes
 - » Déterminer les coûts des solutions
 - » Effectuer des mesures sur les algorithmes
-

Comme nous l'avons vu au [Chapitre 1](#), un algorithme est constitué d'une série d'étapes et sert à résoudre un problème.

L'entrée de données constitue dans la plupart des cas la base de la résolution de ce problème, et elle présente parfois des contraintes dont il importe de tenir compte dans toute solution avant de pouvoir considérer que l'algorithme fonctionne bien. La première section de ce chapitre vous montre comment envisager la *solution du problème* (il s'agit de la solution du problème que vous voulez résoudre). Elle vous permet de comprendre pourquoi il est nécessaire de concevoir des algorithmes qui soient à la fois flexibles (c'est-à-dire capables de prendre en compte un vaste ensemble de données entrantes) et efficaces (c'est-à-dire produisant le résultat désiré).

Certains problèmes sont vraiment compliqués. En les examinant, vous pouvez décider qu'ils sont trop compliqués pour être résolus. Se sentir dépassé par un problème est chose courante. Le plus souvent, pour résoudre la difficulté, il suffit de diviser le problème en éléments plus petits, chaque élément pouvant être géré de façon indépendante. L'approche de type « diviser pour régner », appliquée à la résolution des problèmes et étudiée dans la deuxième section de ce chapitre, faisait initialement référence au domaine de la guerre.



La troisième section de ce chapitre fait référence à l'approche gloutonne de la résolution des problèmes. Le mot « glouton » est normalement chargé d'une connotation négative, mais ce ne sera pas le cas ici. Un *algorithme glouton* est un algorithme consistant à faire un choix optimum à chaque étape. L'objectif est d'obtenir une solution globalement optimale au problème. Cette stratégie n'est malheureusement pas toujours probante, mais elle vaut toujours la peine d'être essayée. Elle aboutit souvent à une solution *assez satisfaisante*, et constitue donc un bon point de départ.

Quelle que soit l'approche que vous choisirez pour résoudre un problème, tout algorithme a un coût. Ceux qui ont grand besoin des algorithmes, en tant que bons clients, veulent obtenir les meilleures conditions, ce qui signifie qu'ils procèdent à une analyse coût/bénéfice. Naturellement, obtenir les meilleures conditions suppose aussi que l'utilisateur ait une idée de ce qui serait une solution acceptable. Obtenir une solution trop précise ou trop riche est souvent inutile. Par conséquent, pour contrôler les coûts, il faut obtenir le nécessaire en termes de résultat, mais rien de plus.

Pour savoir ce qu'un algorithme peut vous donner, il faut que vous sachiez l'évaluer de différentes manières. Il s'agit d'acquérir une vision de son utilité, de sa dimension, des ressources mobilisées et du coût. Plus important, l'évaluation permet de procéder à des comparaisons. Sans mesures d'évaluation, vous ne pourriez pas comparer les algorithmes, et faute de pouvoir les

comparer, vous ne pourriez pas choisir le meilleur pour une tâche donnée.

Commencer à résoudre un problème

Avant de pouvoir résoudre un problème quelconque, il faut que vous le compreniez. Il ne s'agit pas simplement d'en prendre la mesure. Savoir de quels intrants vous disposez et de quels extrants vous avez besoin est un bon point de départ, mais cela ne saurait suffire pour aboutir à une solution. Le processus de résolution consiste, entre autres :



- » à trouver comment d'autres ont abouti à de nouvelles solutions ;
- » de quelles ressources on dispose ;
- » à déterminer les solutions qui convenaient dans le passé pour des problèmes similaires ;
- » à déterminer les solutions qui n'ont pas produit un résultat désirable.

Les sections qui suivent vous permettent de comprendre ces phases de la résolution d'un problème. Rendez-vous compte que vous ne procéderez pas nécessairement à ces phases dans l'ordre, et qu'il vous arrivera de revenir sur une phase après avoir obtenu davantage d'informations. Le processus de résolution est itératif : vous le poursuivez jusqu'à avoir acquis une bonne compréhension du problème.

Modéliser les problèmes du monde réel

Les problèmes du monde réel diffèrent de ceux que l'on trouve dans les manuels. Souvent l'auteur d'un manuel invente un exemple simple pour permettre au lecteur de mieux comprendre les principes fondamentaux en jeu. Cet exemple ne reflète qu'un aspect d'un problème en réalité plus complexe. Lorsque l'on veut résoudre un problème dans le monde réel, il faut souvent combiner plusieurs techniques en vue d'élaborer une solution complète. Ainsi, par exemple, pour déterminer la meilleure réponse à un problème, vous pourriez :

- 1. Avoir besoin de trier les réponses selon un critère spécifique.**
- 2. Effectuer un filtrage et une transformation.**
- 3. Rechercher le résultat.**

Sans cette succession d'étapes, comparer les réponses de façon adéquate risque de se révéler impossible, et l'on aboutit à un résultat qui laisse à désirer. Une série d'algorithmes utilisés ensemble pour obtenir le résultat désiré s'appelle un *ensemble*. À propos de l'utilisation d'un ensemble dans le domaine de l'apprentissage machine.

Néanmoins, les problèmes qui se posent dans le monde réel sont plus complexes encore que le simple examen de données statiques ou une itération unique sur ces données. Ainsi, par exemple, tout ce qui se déplace, comme une auto, un avion ou un robot, reçoit constamment un input. Or, chaque input mis à jour inclut une information d'erreur qu'une solution réelle devra intégrer dans le résultat pour que ces machines continuent à fonctionner correctement. Les calculs permanents supposent non seulement des algorithmes supplémentaires, mais également l'algorithme proportionnel intégral dérivé (PID – pour une explication détaillée de cet algorithme, voir <http://www.ni.com/white-paper/3782/en/>) afin de contrôler la machine à l'aide d'une boucle de rétroaction. Chaque calcul rend plus précise la solution utilisée pour contrôler la machine, et c'est la raison pour laquelle une machine, quand on l'utilise pour la première fois, doit souvent passer par une phase

de réglage (si vous avez l'habitude de vous servir d'un ordinateur, la notion d'itérations vous est peut-être familière, mais les PID concernent les systèmes continus, avec lesquels il n'y a pas d'itérations). La recherche de la solution adéquate correspond à ce que l'on appelle le *temps de stabilisation* : c'est le temps pendant lequel l'algorithme qui contrôle la machine n'a pas encore trouvé la bonne réponse.

Dans la modélisation d'un problème du monde réel, il faut aussi tenir compte des problèmes non évidents qui surgissent. Une solution évidente, même si elle se fonde sur un apport mathématique significatif et sur une théorie solide, ne sera pas nécessairement viable. Durant la Seconde Guerre mondiale, par exemple, les Alliés ont été confrontés à un grave problème, celui des pertes de bombardiers. Des ingénieurs ont donc analysé chaque impact de balle dans tous les avions qui revenaient. À l'issue de l'analyse, les ingénieurs ont opté pour un blindage plus lourd des avions. Cette solution n'a pas marché. C'est alors qu'un mathématicien, Abraham Wald, a suggéré une solution non évidente : placer des plaques de blindage partout où il n'y avait aucun impact de balle (considérant que les parties de la carlingue qui avaient été trouées étaient déjà suffisamment résistantes, car sinon l'avion ne serait pas revenu). Cette solution s'est révélée bonne et elle est aujourd'hui utilisée pour la prise en compte du *biais du survivant* (le fait que les survivants d'une catastrophe, souvent, ne présentent pas la propriété qui a été la véritable cause de la perte) en algorithmique. Pour plus de détails sur cette anecdote historique fascinante, consultez la page <http://www.macgetit.com/solving-problems-of-wwii-bombers/>. En résumé, les biais et autres complications que l'on rencontre dans les problèmes de modélisation peuvent aboutir à des solutions inopérantes.

La modélisation dans le monde réel peut aussi prendre en compte ce que les scientifiques considèrent normalement comme des propriétés indésirables. Le bruit, par exemple, est souvent considéré comme indésirable parce qu'il masque les données sous-jacentes. Songeons à l'aide auditive qui supprime le bruit

pour que le sujet entende mieux (pour plus de détails, voir l'étude sur la page <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4111515/>). Il existe diverses méthodes pour supprimer le bruit, et vous en trouverez dans ce livre, notamment au [Chapitre 9](#), à propos d'un autre sujet. Cependant, aussi contre-intuitif que cela puisse paraître, l'ajout de bruit aussi nécessite un algorithme produisant un résultat exploitable. En 1983, par exemple, Ken Perlin, voulant éviter le look « machinique » des images produites par ordinateur, a développé un algorithme à cet effet. C'est ainsi qu'est apparu ce que l'on a appelé le bruit de Perlin. L'utilité de ce résultat est telle que Ken Perlin a reçu un prix en récompense pour ses travaux (voir <http://mrl.nyu.edu/~perlin/doc/oscar.html>). D'autres, comme Steven Worley, ont créé d'autres sortes de bruits pour modifier les images d'une autre manière (voir sur la page <http://procworld.blogspot.com/2011/05/hello-worley.html>, une comparaison entre le bruit de Perlin et le bruit de Worley). L'idée est que le besoin de supprimer ou d'ajouter du bruit dépend du type de problème à résoudre. En situation réelle, il faut souvent faire des choix qui peuvent ne pas être évidents en laboratoire ou au cours du processus d'apprentissage.



L'idée générale, ici, est qu'il faut souvent plusieurs itérations pour trouver des solutions, qu'il faut parfois consacrer beaucoup de temps à affiner celles-ci, et que les solutions évidentes peuvent ne pas être applicables du tout. Quand on modélise un problème du monde réel, on commence par essayer les solutions proposées dans les manuels, mais il faut ensuite aller au-delà de la théorie pour pouvoir trouver la véritable solution du problème. En lisant ce livre, vous découvrirez une grande variété d'algorithmes, lesquels vous aideront tous à trouver des solutions. Ce qu'il importe de retenir, c'est que vous aurez parfois besoin de combiner ces exemples de différentes façons et de trouver des méthodes pour interagir avec les données de manière à déterminer des caractéristiques qui correspondent au résultat que vous voulez obtenir.

Trouver des solutions et des contre-exemples

La section précédente est une introduction aux aléas de la recherche de solutions dans le monde réel, laquelle comporte des aspects que les solutions trouvées en laboratoire ne peuvent pas prendre en compte. Cependant, trouver une solution – même si c’est une bonne solution – ne suffit pas, car même les bonnes solutions peuvent parfois conduire à un échec. Se faire l’avocat du diable en mettant en évidence des contre-exemples constitue une étape importante dans le processus de résolution d’un problème. L’intérêt des contre-exemples est le suivant :

- » Ils permettent éventuellement de rejeter la solution.
- » Ils permettent de mieux définir la solution en l’encadrant dans des limites.
- » Ils permettent d’étudier des situations dans lesquelles l’hypothèse sur laquelle s’appuie la solution demeure non testée.
- » Ils permettent d’appréhender les limites de la solution.

Un exemple courant de solution accompagnée d’un contre-exemple est le problème posé par la proposition « tous les nombres premiers sont impairs » (les nombres premiers étant les entiers divisibles seulement par eux-mêmes et par 1 pour l’obtention d’un résultat entier). Bien sûr, le nombre 2 est un nombre premier et il n’est pas impair, par conséquent la proposition initiale est fausse. On peut alors la nuancer en déclarant que tous les nombres premiers sont impairs, sauf le nombre 2. La solution partielle au problème de la détermination de l’ensemble des nombres premiers consiste à retenir les nombres impairs, le nombre 2 faisant exception puisqu’il est pair. À cette deuxième étape, rejeter la solution n’est plus possible, mais le rectificatif apporté à la proposition initiale fournit une limite.

En jetant le doute sur la proposition initiale, vous pouvez aussi étudier les situations dans lesquelles l'hypothèse selon laquelle tous les nombres premiers sauf 2 sont impairs pourrait se révéler fausse. Le nombre 1, par exemple, est impair mais n'est pas considéré comme un nombre premier (pour plus de détails, voir <https://primes.utm.edu/notes/faq/one.html>). La proposition initiale est maintenant complétée par deux limites, et doit être reformulée ainsi : les nombres premiers sont plus grands que 1 et généralement impairs, sauf 2 qui est pair. Les limites au domaine des nombres premiers sont mieux définies en identifiant et en prenant en compte les contre-exemples. Au passage, 0 n'est pas non plus considéré comme un nombre premier, pour les raisons énoncées sur la page <http://math.stackexchange.com/questions/539174/is-zero-a-prime-number>.



Quand le problème devient plus complexe, les possibilités de trouver des contre-exemples augmentent. Une règle essentielle est que, à l'instar de la fiabilité, davantage de points de défaillance impliquent davantage de possibilités d'échec. Il est important d'envisager l'algorithmique sous cet angle. Des ensembles constitués d'algorithmes simples peuvent produire de meilleurs résultats, et avec moins de possibilités de contre-exemples, qu'un unique algorithme complexe.

Sur les épaules des géants



Il est un mythe qui défie toute explication, que les techniques actuellement utilisées pour traiter des quantités considérables de données seraient des nouveautés. Certes, de nouveaux algorithmes sont mis au point continuellement, mais ces algorithmes se fondent sur tous ceux qui ont été développés précédemment. Nous considérons Isaac Newton comme un fameux découvreur, or Newton lui-même avait déclaré : « Si j'ai vu plus loin, c'est parce que je m'étais juché sur les épaules de géants » (pour un complément d'informations et pour d'autres citations, voir https://fr.wikipedia.org/wiki/Isaac_Newton).

Le fait est que les algorithmes utilisés aujourd'hui n'étaient même pas une nouveauté au temps d'Aristote (à propos des mathématiques d'Aristote, voir <http://plato.stanford.edu/entries/aristotle-mathematics/>) et de Platon (à propos des mathématiques de Platon, voir http://www.storyofmathematics.com/greek_plato.html). Les origines des algorithmes que nous utilisons aujourd'hui remontent si loin dans l'Histoire que tout ce que l'on peut affirmer, c'est que les mathématiques reposent sur des adaptations des connaissances des temps anciens. Le fait que les algorithmes remontent à l'Antiquité devrait nous rassurer, dans la mesure où les algorithmes utilisés de nos jours se fondent sur des connaissances qui sont éprouvées depuis plusieurs millénaires.

Il ne faudrait pas croire pour autant que les mathématiciens n'ont jamais changé le cours de choses. La théorie de John Nash, par exemple, c'est-à-dire l'Équilibre de Nash, a représenté un changement notable dans la vision de l'économie (pour un tutoriel sur cette théorie, voir <https://www.khanacademy.org/economics-finance-domain/microeconomics/nash-equilibrium/tutorial>). Naturellement, la reconnaissance de ces travaux est lente (et parfois elle n'a jamais lieu). Nash a dû attendre longtemps avant d'être reconnu par ses pairs (voir <https://www.princeton.edu/main/news/archive/S42/72/29C63/index> même s'il s'est vu décerner le prix Nobel d'économie pour ses contributions. Pour l'anecdote, un film raconte l'histoire de John Nash : ce film, *Un homme d'exception*, comporte des scènes controversées, notamment celle dans laquelle il est dit que l'Équilibre de Nash infirme en partie les travaux d'Adam Smith, un auteur à qui l'on doit également des théories économiques (voir notamment une discussion sur la page <https://www.quora.com/Was-Adam-Smith-wrong-as-claimed-by-John-Nash-in-the-movie-A-Beautiful-Mind>).

Diviser pour régner

Si les problèmes étaient faciles à résoudre, tout le monde les résoudrait. Or, le monde est rempli de problèmes irrésolus et cela ne risque pas de changer avant longtemps, pour une raison simple : les problèmes paraissent souvent si considérables qu'aucune solution n'est imaginable. Les guerriers des temps anciens étaient confrontés à une situation similaire. L'armée ennemie semblait parfois si vaste, comparée à leurs forces si limitées, que gagner la guerre était un problème extrêmement difficile, voire impossible à résoudre. Et cependant, en divisant l'armée ennemie en petits groupes et en attaquant ces groupes un par un, une petite armée pouvait parfois vaincre un adversaire bien plus imposant (les anciens Grecs, les Romains et Napoléon Bonaparte ont très bien su recourir à cette stratégie, consistant à diviser pour vaincre – ou selon la formule consacrée, pour régner : pour plus de détails, lire *Napoléon pour les Nuls*, de J. David Markham).

Nous sommes confrontés au même problème que ces guerriers du passé. Souvent, les ressources dont nous disposons semblent très limitées et inadéquates. Pourtant, en divisant un problème considérable en petits éléments bien plus faciles à appréhender, on peut aboutir à une solution qui fonctionne pour l'ensemble du problème. Ce principe est à la base même des algorithmes : procéder par étapes et résoudre les problèmes morceau par morceau. Les sections qui suivent expliquent en détail cette approche de la résolution des problèmes.

Éviter la recherche de solutions par force brute

La recherche par force brute, ou recherche exhaustive, consiste à essayer une par une toutes les réponses possibles jusqu'à ce que l'on ait trouvé la meilleure qui soit. C'est certes une méthode rigoureuse, indiscutablement, mais c'est aussi, dans la plupart des cas, un gaspillage de temps et de ressources. Tester toutes les réponses, même lorsqu'il est facile de prouver qu'une réponse

particulière n'a aucune chance d'être la bonne, c'est gaspiller un temps qu'un algorithme pourrait utiliser pour traiter des réponses ayant plus de chances de succès. En outre, tester les différentes réponses par cette méthode entraîne généralement un gaspillage de ressources comme la mémoire. Songez que pour trouver la combinaison d'un cadenas, la méthode par force brute consisterait à tester tout d'abord la combinaison 0, 0, 0 tout en sachant qu'il n'y a aucune chance pour que ce soit la bonne, compte tenu des caractéristiques physiques des cadenas à combinaison, puis à tester la combinaison 0, 0, 1, ce qui serait tout aussi ridicule.



Il est important de comprendre que chaque type de solution a ses avantages, parfois tout petits. C'est le cas d'une solution par la force brute : sachant que l'on teste chaque réponse, aucun prétraitement n'est nécessaire. Néanmoins, le temps économisé en évitant le prétraitement a peu de chances de compenser le temps perdu à essayer toutes les réponses. Opter pour la solution de la force brute peut cependant se justifier dans les cas suivants :

- » Quand il est essentiel de trouver une solution, dans la mesure où elle existe.
- » Quand la dimension du problème est limitée.
- » Quand il est possible de recourir à l'heuristique pour réduire le nombre de solutions.
- » Quand la simplicité de la procédure est plus importante que sa rapidité.

Commencer par simplifier

La solution par force brute présente un grave inconvénient, celui d'attaquer le problème tout entier. C'est un peu comme si, pour trouver un ouvrage dans une bibliothèque, on examinait les livres un par un sur une première étagère, puis sur une deuxième et ainsi de suite, sans jamais envisager une méthode qui simplifierait la recherche. Au contraire, avec l'approche de type « diviser pour régner », on commencerait par distinguer les rayons des livres

pour adultes et les rayons des livres pour enfants. Ensuite, ayant retenu les rayons pour adultes, on y distinguerait les différentes catégories, et enfin, on limiterait la recherche à la catégorie à laquelle appartient l'ouvrage désiré. C'est le principe de systèmes de classification comme la classification décimale de Dewey (voir https://fr.wikipedia.org/wiki/Classification_d%C3%A9cimale_de). Cette approche simplifie le problème. En réduisant le nombre d'items candidats, la tâche devient plus rapide et plus facile.

Dans l'approche « diviser pour régner », diviser est aussi un moyen essentiel de mieux comprendre le problème. Il peut s'avérer difficile de tenter de saisir l'organisation d'un « package » pris dans son intégralité. Quand vous savez que l'ouvrage de psychologie comparative que vous recherchez se trouve dans la subdivision 156 de la division 150, laquelle appartient à la classe 100, la tâche est plus facile. Ce n'est plus qu'un petit problème, sachant que tous les livres de la subdivision 156 traitent du sujet qui vous intéresse. L'algorithmique applique le même principe. En simplifiant le problème, on peut définir une série d'étapes simples menant à la solution. On réduit ainsi le temps et la quantité de ressources nécessaires pour trouver la solution, et l'on accroît ses chances de trouver précisément la solution dont on a besoin.

Il est généralement préférable de décomposer le problème

Après avoir divisé le problème en éléments gérables, il s'agit de venir à bout de chaque élément. Pour cela, il est nécessaire d'avoir défini précisément le problème. Vous ne recherchez pas n'importe quel ouvrage de psychologie comparative, vous voulez un ouvrage dont l'auteur est George Romanes. Vous savez que l'ouvrage désiré se trouve dans la subdivision 156 de la classification décimale de Dewey et c'est un bon début, mais le problème n'est pas résolu pour autant. Il vous faut maintenant un processus pour passer en revue tous les ouvrages de la subdivision 156, en vue de

trouver celui qui vous intéresse. Vous pourriez envisager une étape supplémentaire qui consisterait à rechercher les ouvrages traitant spécifiquement d'un sujet donné. Pour que ce processus soit viable, il serait nécessaire de décomposer le problème de façon totale, de définir précisément votre besoin, puis, après avoir acquis une perception approfondie du problème, d'exécuter la série d'étapes appropriée (l'algorithme) pour aboutir au bon résultat.

AVEC LES ALGORITHMES, IL N'Y A PAS DE VÉRITÉ ABSOLUE

Peut-être pensez-vous que vous pourriez définir un scénario dans lequel vous utiliseriez toujours un certain type d'algorithme pour résoudre un type de problème particulier. Or, ce n'est pas le cas. Les mérites relatifs des techniques de recherche par force brute et de l'approche « diviser pour régner » pour résoudre certains problèmes font l'objet de débats. Il n'est pas surprenant de constater que l'approche « diviser pour régner » n'est pas préférée dans toutes les situations. Ainsi, par exemple, s'il s'agit de rechercher la plus forte valeur parmi les valeurs non triées contenues dans un tableau, la recherche par force brute sera sans doute la meilleure option. Pour une étude sur ce sujet, voir le lien <http://stackoverflow.com/questions/11043226/why-do-divide-andconquer-algorithms-often-run-faster-than-brute-force>.

Fait intéressant, la recherche par force brute est aussi la technique la plus économe en ressources dans ce cas particulier. Il ne faut jamais oublier que les règles ont des exceptions et que la connaissance de ces exceptions permet d'économiser du temps et des efforts par la suite.

La gloutonnerie n'est pas toujours un vilain défaut

Dans certains cas, on ne voit pas la fin du processus de résolution, et parfois même, on ignore si l'on aboutira à un résultat. Ce dont on peut s'assurer, c'est de la possibilité de remporter des victoires partielles dans la recherche de la solution, en espérant aussi une victoire finale. C'est de cette approche que procède la méthode de l'algorithme glouton, qui consiste à rechercher une solution globale en sélectionnant le meilleur résultat possible à chaque étape de la résolution du problème.



On pourrait penser que gagner toutes les batailles signifie nécessairement gagner la guerre, mais les choses ne se passent pas toujours ainsi dans le monde réel. On parle de *victoire à la Pyrrhus* lorsque celui qui a gagné toutes les batailles finit tout de même perdant parce que le coût de la victoire dépasse significativement la somme des gains. Pour découvrir cinq exemples de victoires à la Pyrrhus, consultez la page <http://www.history.com/news/history-lists/5-famous-Pyrrhic-victories> La plus importante leçon à tirer de ces exemples est qu'un algorithme glouton est souvent la méthode qui fonctionne, mais pas toujours, et qu'il faut donc envisager la meilleure solution globale du problème plutôt que de se laisser aveugler par des gains intermédiaires. Les sections qui suivent expliquent comment éviter une victoire à la Pyrrhus quand on utilise des algorithmes.

Appliquer la logique de l'algorithme glouton

Il arrive souvent que la logique de l'algorithme glouton intervienne dans un processus d'optimisation. L'algorithme traite le problème étape par étape et à chaque moment, il ne prend en compte que l'étape en cours. Tout algorithme glouton repose sur les deux hypothèses suivantes :

- » À chaque étape correspond un unique choix optimal.

- » En faisant le choix optimal à chaque étape, on aboutit à la solution optimale du problème global.

On peut trouver de nombreux exemples d'algorithmes gloutons, toujours optimisés pour l'exécution d'une tâche particulière. Voici quelques exemples courants d'algorithmes gloutons utilisés pour l'analyse graphique (pour plus de détails sur les graphes, voir [Chapitre 9](#)) et pour la compression de données (pour plus de détails sur la compression de données, voir [Chapitre 14](#)), avec la raison de leur utilisation :

- » **L'arbre couvrant de poids minimum (ARPM), de Kruskal** : Cet algorithme démontre un principe des algorithmes gloutons auquel on ne pense pas immédiatement. En l'occurrence, cet algorithme sélectionne, parmi les chemins entre deux sommets, celui qui représente la plus petite valeur, et non la plus grande valeur comme le mot glouton pourrait initialement le suggérer. Grâce à ce type d'algorithme, on peut trouver le plus court chemin entre deux points sur une carte ou exécuter d'autres tâches liées à une représentation graphique.
- » **L'arbre couvrant minimal de Prim** : Cet algorithme divise en deux parties un graphe non orienté (chaque arête relie deux sommets, ou nœuds, entre lesquels il n'est pas tenu compte d'un sens de parcours). Il sélectionne les sommets de telle sorte que le poids total des deux parties du graphe soit le moins élevé possible. Cet algorithme est utilisé, par exemple, pour déterminer la distance la plus courte entre le point de départ et le point d'arrivée d'un labyrinthe.
- » **Le codage de Huffman** : Cet algorithme est bien connu des informaticiens, car il est à la base de diverses techniques de compression de données. Il assigne un code à chaque élément du flux entrant de données, de telle sorte que les données les plus utilisées se voient attribuer le code le plus court. Lors du processus de compression d'un texte,

par exemple, le code le plus court sera normalement attribué à la lettre E, car c'est la lettre la plus utilisée de tout l'alphabet. Ainsi, en modifiant la technique de codage, il est possible de réduire nettement la taille d'un texte ou d'une autre série de données, et de ce fait, le temps de transmission.

Parvenir à une bonne solution

Parce que les scientifiques et les mathématiciens utilisent très souvent des algorithmes gloutons, le [Chapitre 15](#) traite ce sujet en détail. Toutefois, il est important de se rendre compte que ce que l'on veut réellement, c'est une bonne solution, et non pas simplement une solution particulière. Dans la plupart de cas, une bonne solution donne des résultats optimaux dont on peut prendre la mesure, mais le mot « bonne » peut avoir plusieurs sens, selon le problème étudié. Il convient de se demander quel problème il s'agit de résoudre et quelle solution permet de le résoudre de la façon la plus adaptée aux besoins. Dans le domaine de l'ingénierie, par exemple, vous pourriez être amené à évaluer des solutions en tenant compte de la masse, de la taille, du coût, *etc.*, ou peut-être d'une combinaison de toutes ces grandeurs qui satisfasse à certaines spécifications.

Pour situer le problème dans son contexte, supposons que vous fabriquiez une machine destinée à rendre la monnaie sous forme du plus petit nombre possible de pièces (par exemple, pour la caisse automatique d'un magasin). S'il s'agit d'opter pour le plus petit nombre possible de pièces, c'est afin de limiter l'usure de la machine, le poids de la réserve de pièces à constituer et le temps nécessaire pour exécuter l'opération (les consommateurs étant toujours pressés). Avec un algorithme glouton, on peut résoudre le problème en utilisant en priorité les plus grosses pièces. Pour un rendu de monnaie de 0,16 euro, par exemple, on utilisera une pièce de 10 centimes, une pièce de 5 centimes et une pièce de 1 centime.



Un problème survient lorsque la machine ne peut plus utiliser n'importe quelle pièce pour produire la solution. Supposons, par exemple, qu'elle soit à court de pièces de 10 et de 5 centimes. Pour rendre 0,80 euro, l'algorithme sélectionnera d'abord une pièce de 50 centimes, puis une pièce de 20 centimes. Faute de disposer de pièces de 10 et de 5 centimes, l'algorithme y ajoutera 5 pièces de 2 centimes, ce qui fera au total sept pièces. Or, la solution optimale pour un rendu de 0,80 euro aurait consisté à sélectionner plutôt quatre pièces de 20 centimes. Ainsi donc, l'algorithme glouton fournit une solution particulière, mais ce n'est pas la bonne solution (la solution optimale) dans cet exemple. Le problème du rendu de monnaie fait l'objet d'une attention considérable, car il est difficile à résoudre. Pour plus de détails, vous pouvez lire par exemple *Combinatorics of the Change-Making Problem* d'Anna et Michal Adamaszek (voir <http://www.sciencedirect.com/science/article/pii/S0195669809001> ou bien Wikipédia (https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_rendu_de_monnaie

Calculer les coûts et suivre une heuristique

Même quand on trouve une bonne solution, c'est-à-dire une solution à la fois viable et performante, il reste nécessaire d'en connaître précisément le coût. Il arrive que le coût d'utilisation d'une solution donnée soit trop élevé, même en tenant compte de tout le reste. La réponse peut tarder un tout petit peu trop, ou consommer une quantité considérable de ressources. La recherche d'une bonne solution passe par la création d'un contexte dans lequel on pourra tester entièrement l'algorithme, la situation qu'il engendre, les opérateurs qu'il utilise pour effectuer ces changements et le temps nécessaire pour obtenir une solution.

Souvent, il s'avère qu'une *approche heuristique*, c'est-à-dire reposant sur l'autodécouverte et produisant des résultats suffisamment exploitables (pas nécessairement optimaux, mais

acceptables) est la méthode qu'il faut pour résoudre un problème. Quand un algorithme exécute pour vous certaines tâches requises, vous économisez du temps et des efforts, car vous êtes en mesure de mettre au point des algorithmes qui décèlent les tendances mieux que ne le font les humains. L'autodécouverte est donc le processus qui permet à l'algorithme de vous indiquer une voie potentiellement utile vers une solution (mais il faut encore compter sur l'intuition et l'entendement humains pour savoir si cette solution est la bonne). Les sections qui suivent décrivent des techniques que vous pouvez utiliser pour calculer le coût d'un algorithme en recourant à l'heuristique comme méthode de détermination de l'utilité réelle d'une solution donnée.

Représenter le problème comme un espace

On appelle *espace-problème* l'environnement dans lequel s'effectue la recherche d'une solution. L'espace-problème est constitué d'une série d'états et des opérateurs utilisés pour changer ces états. Considérons, par exemple, un jeu de placement constitué de huit tuiles à disposer sur une grille de 353 cases. Chaque tuile fait apparaître une partie de l'image, et les tuiles sont initialement disposées de façon aléatoire, formant une image brouillée. Le but du jeu est de déplacer une tuile à la fois de manière à placer les tuiles dans le bon ordre pour révéler l'image. Vous pouvez voir un exemple de ce type de puzzle sur la page <http://mypuzzle.org/sliding>.

L'état initial, la disposition aléatoire des tuiles et l'état représentant le but à atteindre – les tuiles dans un ordre particulier – constituent ensemble *l'instance du problème*. Le problème peut être représenté graphiquement à l'aide d'un *graphe espace-problème*. Chaque sommet (ou nœud) du graphe espace-problème représente un état (les huit tuiles dans une disposition particulière). Les arêtes représentent les opérations, par exemple le déplacement de la tuile numéro huit vers le haut. Quand on

déplace une tuile vers le haut, l'image est modifiée : elle passe à un autre état.

Gagner la partie en passant de l'état initial à l'état final n'est pas l'unique considération. Pour résoudre le problème de façon efficace, encore faut-il que la tâche soit exécutée avec le plus petit nombre possible de coups, autrement dit, en utilisant le plus petit nombre possible d'opérateurs. Le nombre minimum d'opérations pour résoudre le problème est ce que l'on appelle la *dimension du problème*.

Dans la représentation spatiale d'un problème, plusieurs facteurs doivent entrer en ligne de compte. Ainsi, par exemple, il faut tenir compte du nombre maximum de sommets pouvant être stockés en mémoire, et qui représente la *complexité de l'espace*. Quand la mémoire est insuffisante pour stocker tous les sommets en même temps, l'ordinateur doit stocker une partie des sommets ailleurs, par exemple sur le disque dur, ce qui peut ralentir considérablement l'algorithme. Pour savoir si les sommets pourront tenir en mémoire, il faut tenir compte de la *complexité en temps*, qui est le nombre maximum de sommets créés pour résoudre le problème. Par ailleurs, il importe de prendre en compte le *facteur de ramification*, c'est-à-dire le nombre moyen de sommets créés dans le graphe espace-problème pour résoudre le problème.

Compter sur le hasard et avoir de la chance

Il est possible de résoudre un problème de recherche en utilisant des techniques de force brute (voir « Éviter la recherche de solutions par force brute », précédemment dans ce chapitre). L'avantage de cette approche est qu'on n'a pas besoin d'avoir des connaissances particulières dans un domaine pour pouvoir utiliser un de ces algorithmes. Un algorithme utilise généralement l'approche la plus simple possible. L'inconvénient est que

l'approche par la force brute n'est pertinente que pour un petit nombre de sommets. Voici quelques exemples courants d'algorithmes de recherche par force brute :

- » **Parcours en largeur** : Cette technique consiste à partir du sommet racine (ou nœud racine), à explorer tout d'abord chaque nœud fils, puis à se placer au niveau suivant et à progresser niveau par niveau jusqu'à trouver une solution. L'inconvénient de cet algorithme est qu'il doit stocker en mémoire tous les nœuds. Par conséquent, si le nombre de nœuds est élevé, l'algorithme devra utiliser une quantité de mémoire considérable. En revanche, cette technique permet de repérer les nœuds redondants, et ainsi, de gagner du temps, et elle aboutit toujours à une solution.
- » **Parcours en profondeur** : Cette technique consiste à partir du sommet racine et à explorer une série de nœuds fils connectés jusqu'à ce que l'on atteigne un nœud externe. La progression se fait branche par branche jusqu'à trouver une solution. L'inconvénient de cet algorithme est qu'il ne peut pas repérer les nœuds redondants, et qu'il risque donc de parcourir le même chemin plus d'une fois. Cet algorithme peut aussi ne pas aboutir du tout à une solution, par conséquent il est indispensable de définir un seuil de coupure afin d'éviter que l'algorithme poursuive la recherche indéfiniment. Un avantage de cette approche est qu'elle économise la mémoire.
- » **Parcours bidirectionnel** : Cette technique consiste à effectuer la recherche simultanément depuis le nœud racine et depuis le nœud final jusqu'à ce que les deux parcours se rejoignent au milieu du graphe. Un avantage de cette méthode est qu'elle fait économiser du temps, car elle permet de trouver la solution plus vite que bon nombre d'autres méthodes de type force brute. En outre, elle utilise la mémoire de façon plus efficiente que les autres méthodes et elle aboutit toujours à une solution. Son principal inconvénient est la complexité de son application, qui se traduit par un cycle de développement plus long.

Utiliser une heuristique et une fonction de coût

Certains trouvent le mot *heuristique* trop compliqué. Il serait tout aussi facile de dire que l'algorithme procède à une estimation éclairée, puis à un nouvel essai en cas d'échec. Contrairement aux méthodes de force brute, les algorithmes heuristiques apprennent. Ils utilisent aussi des fonctions de coût pour faire de meilleurs choix. En conséquence, ils sont plus compliqués, mais ils présentent un avantage certain pour la résolution des problèmes complexes. À l'instar des algorithmes de force brute, les algorithmes heuristiques sont très variés et chaque type présente sa propre série d'avantages, d'inconvénients et d'exigences particulières. La liste suivante expose quelques algorithmes heuristiques parmi les plus courants :

» **L'algorithme de recherche purement heuristique :**

L'algorithme développe les nœuds par ordre de coût. Il gère deux listes, une liste fermée comportant les nœuds déjà explorés et une liste ouverte comportant les nœuds qu'il lui reste à explorer. À chaque itération, l'algorithme développe le nœud correspondant au coût le moins élevé possible. Il place tous les nœuds fils dans la liste fermée et calcule les coûts qui leur sont associés. Il replace les nœuds fils auxquels est associé un coût peu élevé dans la liste ouverte et supprime les nœuds fils auxquels est associé un coût élevé. Ainsi, pour trouver la solution, l'algorithme effectue une recherche intelligente en se fondant sur les coûts.

» **L'algorithme A* :** Cet algorithme suit le coût des nœuds à mesure qu'il les explore en utilisant l'équation : $f(n) = g(n) + h(n)$, où

- n est l'identifiant du nœud.
- $g(n)$ est le coût du parcours pour atteindre le nœud n .

- $h(n)$ est le coût estimé pour atteindre le but depuis le nœud n .
- $f(n)$ est le coût estimé du parcours de la racine au but.

L'idée est de rechercher les chemins les plus prometteurs d'abord et d'éviter les chemins coûteux.

» **L'algorithme de recherche best-first** : Cet algorithme choisit toujours le chemin qui se rapproche le plus du but en utilisant l'équation : $f(n) = h(n)$. Il trouve les solutions rapidement, mais peut aussi boucler sans fin, c'est pourquoi il n'est souvent pas considéré comme la méthode optimale pour trouver une solution.

Évaluer les algorithmes

Il est important d'acquérir une connaissance précise de la façon dont fonctionnent les algorithmes, faute de quoi on ne saurait déterminer si un algorithme fonctionne vraiment comme on a besoin qu'il fonctionne. Par ailleurs, sans de bonnes mesures, il n'est pas possible d'effectuer des comparaisons justes afin de savoir si l'on a réellement besoin de trouver une nouvelle méthode de résolution de problème lorsqu'une solution plus ancienne est trop lente ou consomme trop de ressources. Le fait est que la plupart du temps, les algorithmes que vous utiliserez auront été développés par d'autres, même si vous en concevrez éventuellement vous-même. Connaître les bases pour exploiter et comparer différentes solutions puis faire un choix est une compétence essentielle, quand on est amené à utiliser des algorithmes.

La question de l'efficacité intervient dans la découverte et la conception de nouveaux algorithmes depuis que le concept d'algorithme a fait son apparition, raison pour laquelle si souvent, plusieurs algorithmes différents sont en concurrence pour résoudre

le même problème (parfois on a vraiment l'embarras du choix). L'idée de mesurer les caractéristiques des fonctions dans un algorithme et d'analyser le fonctionnement de celui-ci n'est pas nouvelle : déjà en 1843, Ada Lovelace et Charles Babbage avaient étudié les problèmes d'efficacité des algorithmes en référence aux calculateurs (pour un bref historique de la machine de Babbage, voir <http://www.computerhistory.org/babbage/adalovelace/>).

Donald Knuth (<http://www-cs-faculty.stanford.edu/~uno/>), informaticien, mathématicien, professeur émérite à l'université de Stanford et auteur de l'ouvrage monumental *The Art of Computer Programming* (Addison-Wesley), a consacré une grande partie de ses travaux d'études et de recherche à la comparaison d'algorithmes. Il s'est efforcé de formaliser mathématiquement le processus d'estimation des besoins en ressources des algorithmes en vue de pouvoir comparer valablement les solutions. On lui doit la notion d'*analyse d'algorithmes*, un domaine de l'informatique qui est la formalisation du fonctionnement des algorithmes. L'analyse consiste à mesurer les ressources nécessaires en termes de nombre d'opérations que l'algorithme doit exécuter pour aboutir à une solution, ou en termes d'espace occupé (par exemple l'espace de stockage nécessaire dans la mémoire de l'ordinateur).

L'analyse d'algorithme suppose des connaissances en mathématiques et fait appel à des calculs, mais il vous sera très profitable de découvrir, d'apprécier et d'utiliser efficacement des algorithmes. Il s'agit d'un domaine considérablement plus abstrait que tous les autres sujets abordés dans ce livre. Pour que l'étude ne revête pas un aspect trop théorique, les chapitres qui suivent présentent davantage d'aspects pratiques de ces mesures avec l'examen détaillé d'algorithmes. Les sections suivantes vous apportent les éléments de base.

Simuler l'utilisation de machines abstraites

Plus un algorithme doit effectuer d'opérations, plus il est complexe. La complexité est une mesure de l'efficacité d'un algorithme en termes d'utilisation du temps, sachant que toute opération utilise du temps. Pour un même problème, les algorithmes complexes sont généralement moins avantageux que des algorithmes simples, car ils ont besoin de plus de temps pour être exécutés. N'oublions pas que la vitesse d'exécution change beaucoup de choses, que ce soit dans le secteur médical ou dans la finance, ou lorsqu'il s'agit de faire voler un avion en pilotage automatique, ou d'envoyer une fusée dans l'espace. Mesurer la complexité d'un algorithme est une tâche difficile, mais qui est nécessaire si l'on veut pouvoir exploiter la solution qui convient. La première technique de mesure utilise des machines abstraites comme la Random Access Machine (ou machine RAM).



RAM peut signifier aussi *Random-Access Memory* ou mémoire à accès aléatoire, un terme qui désigne la mémoire interne utilisée par un ordinateur lorsqu'il exécute des programmes. Bien qu'il s'agisse du même acronyme, la machine RAM est quelque chose de totalement différent.

Les *machines abstraites* sont non pas des ordinateurs réels, mais des ordinateurs théoriques, dont le fonctionnement est simulé. Elles servent à étudier la façon dont s'exécute un algorithme sur un ordinateur sans le tester sur une machine réelle, tout en faisant dépendre l'étude du type de matériel à utiliser. Une machine RAM exécute des opérations arithmétiques de base et interagit avec des données en mémoire, et ne fait rien de plus. Chacune de ses opérations consomme un pas de temps (une unité de temps). Quand on procède à une simulation de RAM pour évaluer un algorithme, on compte les pas de temps en appliquant la procédure suivante :

- 1. Compter chaque opération (arithmétique) simple comme un pas de temps.**
- 2. Décomposer les opérations complexes en opérations arithmétiques simples et compter les pas de temps comme indiqué à l'étape 1.**

3. Compter chaque accès aux données de la mémoire comme un pas de temps.

Pour réaliser ce comptage, on écrit une version de l'algorithme en pseudo-code (voir [Chapitre 1](#)) et on suit ces étapes en utilisant du papier et un crayon. Au bout du compte, c'est une méthode simple qui procède d'une idée fondamentale du fonctionnement de l'ordinateur, d'une approximation exploitable permettant de comparer les solutions en faisant abstraction de la puissance et de la vitesse de traitement de la machine et du langage de programmation utilisé.



Recourir à une simulation n'est pas la même chose qu'exécuter un algorithme sur un ordinateur, sachant qu'on utilise un input standard et prédéfini. Les mesures sur ordinateur en réel consistent à exécuter un code et à vérifier le temps que dure cette exécution. L'exécution du code sur l'ordinateur est en fait une *référence*, une autre forme de mesure de l'efficacité avec laquelle on tient aussi compte du contexte de l'application (notamment, du type de matériel utilisé et de la programmation informatique). Cette référence est utile, mais pas assez généralisable. Il suffit de penser, par exemple, à la rapidité d'exécution sur les machines les plus récentes d'un algorithme dont l'exécution prenait un temps fou sur les ordinateurs de la génération précédente.

Pour aller plus loin encore dans l'abstraction

Mesurer une série d'étapes destinées à aboutir à la solution d'un problème n'est pas sans poser des difficultés. Dans la section précédente, il est question du comptage des pas de temps (du nombre d'opérations), mais il est parfois nécessaire de calculer l'espace (notamment, l'espace mémoire consommé par l'algorithme). La question de l'espace se pose quand la résolution du problème est gourmande en ressources. Selon le problème à résoudre, les caractéristiques d'un algorithme en termes de

consommation de ressources qu'il conviendra d'étudier pourront être les suivantes :

- » le temps d'exécution ;
- » la mémoire requise ;
- » l'utilisation du disque dur ;
- » la consommation électrique ;
- » la vitesse de transmission des données à travers un réseau.

Entre certains de ces aspects, il peut exister une relation inverse. Une plus grande vitesse d'exécution, par exemple, impliquera davantage de mémoire utilisée ou davantage de consommation électrique. Non seulement plusieurs configurations en termes d'efficacité peuvent être associées à l'exécution d'un algorithme, mais il est possible également de jouer sur les caractéristiques du matériel et sur la programmation logicielle pour atteindre les objectifs fixés. Concernant le matériel, la situation ne sera pas la même selon que l'on utilisera un supercalculateur ou un ordinateur polyvalent, et concernant le logiciel, tout peut dépendre de l'application choisie ou du langage utilisé pour écrire l'algorithme. En outre, la qualité des mesures de performance peut dépendre de la quantité et du type de données que l'algorithme devra traiter.

Les simulations de RAM impliquent un décompte du temps car lorsqu'une solution est susceptible d'être appliquée dans des contextes si variés et lorsque l'utilisation des ressources qu'elle suppose dépend de tant de facteurs, il faut trouver un moyen de simplifier les comparaisons afin de les normaliser. Autrement, il n'est pas possible de comparer les alternatives envisageables. Comme c'est le cas bien souvent pour un grand nombre de problèmes, la solution consiste à recourir à une mesure unique et à considérer qu'elle est applicable partout. En l'occurrence, cette mesure est le temps, que l'on assimile au nombre d'opérations, c'est-à-dire à la complexité de l'algorithme.

Une simulation de RAM consiste à rendre l'algorithme indépendant aussi bien du langage de programmation que du type de machine. Cependant, expliquer le fonctionnement d'une simulation de RAM ne va pas de soi. Dans l'analyse d'algorithme, on se propose d'utiliser le nombre d'opérations déterminé dans la simulation de RAM et d'en faire une fonction mathématique exprimant le comportement de l'algorithme en termes de temps, sous forme d'une quantification des étapes ou des opérations requises lorsque le nombre de données à entrer augmente. Si votre algorithme sert à trier des objets, par exemple, vous pouvez exprimer sa complexité à l'aide d'une fonction associant au nombre d'objets traités le nombre d'opérations nécessaires.

Travailler avec des fonctions

En mathématiques, une fonction est simplement un moyen de faire correspondre une réponse à une ou plusieurs entrées. On peut aussi définir une *fonction* comme une transformation de l'entrée (par des opérations mathématiques) aboutissant à une réponse. Pour certaines valeurs de l'entrée (généralement notée x ou n), on obtient la réponse correspondante à l'aide des opérations mathématiques qui définissent la fonction. Ainsi, par exemple, la fonction $f(n) = 2n$ associe au nombre n une réponse qui est ce nombre n multiplié par 2.

Il est logique d'utiliser la taille de l'entrée alors que le temps est devenu une ressource si précieuse, sachant que nous sommes confrontés à des quantités de données de plus en plus grandes. Tout modéliser sous forme de fonctions mathématiques est un peu moins évident, mais une fonction décrivant la façon dont un algorithme fait correspondre une solution à la quantité de données qu'il reçoit est quelque chose que l'on peut analyser sans nécessiter le concours d'un matériel ou d'un logiciel particulier. Par ailleurs, il est facile de comparer la solution avec d'autres solutions, compte tenu de la dimension du problème. L'analyse d'algorithme est véritablement un concept remarquable, car elle

permet de réduire une série complexe d'étapes à une formule mathématique.

Par ailleurs, la plupart du temps, l'analyse d'algorithme ne se soucie même pas de définir la fonction avec exactitude. Ce qu'il s'agit vraiment de faire, c'est comparer une fonction cible à une autre fonction. Les fonctions à comparer proviennent d'une série de fonctions proposées qui semblent peu performantes lorsqu'elles sont mises en regard de l'algorithme cible. Ainsi, il n'est pas nécessaire d'appliquer à des nombres des fonctions plus ou moins complexes, il suffit de se référer à des fonctions connues, simples et préimplémentées. Cela peut sembler grossier, mais c'est plus efficace et cela revient, d'une certaine manière, à classer la performance des algorithmes par catégories plutôt que d'en obtenir une mesure exacte.

L'utilisation de ces fonctions généralisées est ce que l'on appelle la *comparaison asymptotique* et dans ce livre, cette petite série de fonctions (entre parenthèses et précédées d'un O majuscule) sera souvent utilisée pour exprimer la performance des algorithmes. La [Figure 2-1](#) représente l'analyse d'un algorithme. Sa fonction peut être représentée dans un système de coordonnées cartésiennes, la mesure étant effectuée par une simulation de RAM, l'*abscisse* (la coordonnée x) étant la dimension de l'input et l'*ordonnée* (la coordonnée y), le nombre d'opérations qui en résulte. Trois courbes ont été tracées. La dimension de l'input a son importance. Cependant, la qualité aussi est importante (en termes d'ordonnancement, par exemple, classer des données d'entrée déjà presque classées va plus vite). Par la suite, l'analyse fait apparaître une situation défavorable, $f_1(n)$, une situation moyenne, $f_2(n)$ et une situation favorable, $f_3(n)$. Même si la situation moyenne peut nous donner une idée générale, ce dont il faut réellement se préoccuper est la situation défavorable, car des problèmes peuvent survenir quand l'algorithme mouline pour aboutir à une solution. La comparaison asymptotique, au-delà d'une certaine valeur n_0 (le seuil à partir duquel l'entrée est considérée comme étant de grande dimension), aboutit toujours à un plus grand nombre d'opérations, pour un même input, que la fonction correspondant à

la situation défavorable, f_1 . Par conséquent, la comparaison asymptotique constitue une approche plus pessimiste encore que la fonction qui représente l'algorithme, si bien que quelle que soit la qualité de l'input, on peut être sûr que la situation ne saurait être plus défavorable.

Un certain nombre de fonctions possibles peuvent donner des résultats défavorables, mais le choix des fonctions utilisables dans le cas de la comparaison asymptotique est restreint, sachant que l'objectif est de simplifier la mesure de la complexité en proposant une norme. En conséquence, cette section ne présente qu'un petit nombre des fonctions utilisées dans la comparaison asymptotique. La liste qui suit les présente par ordre croissant de complexité :

- » **Complexité constante, $O(1)$** : Le temps est le même, quelle que soit la quantité d'inputs. Au bout du compte, nous avons un nombre constant d'opérations, quelle que soit la longueur de la série de données entrées. Dans la pratique, ce niveau de complexité est très rare.
- » **Complexité logarithmique, $O(\log n)$** : Le nombre d'opérations croît à un taux moins élevé que l'input, si bien que l'algorithme est moins efficient avec des inputs réduits et plus efficient avec des inputs importants. Un algorithme typique de cette classe est celui de la recherche binaire, présenté au [Chapitre 7](#), qui traite de l'organisation et de la recherche des données.

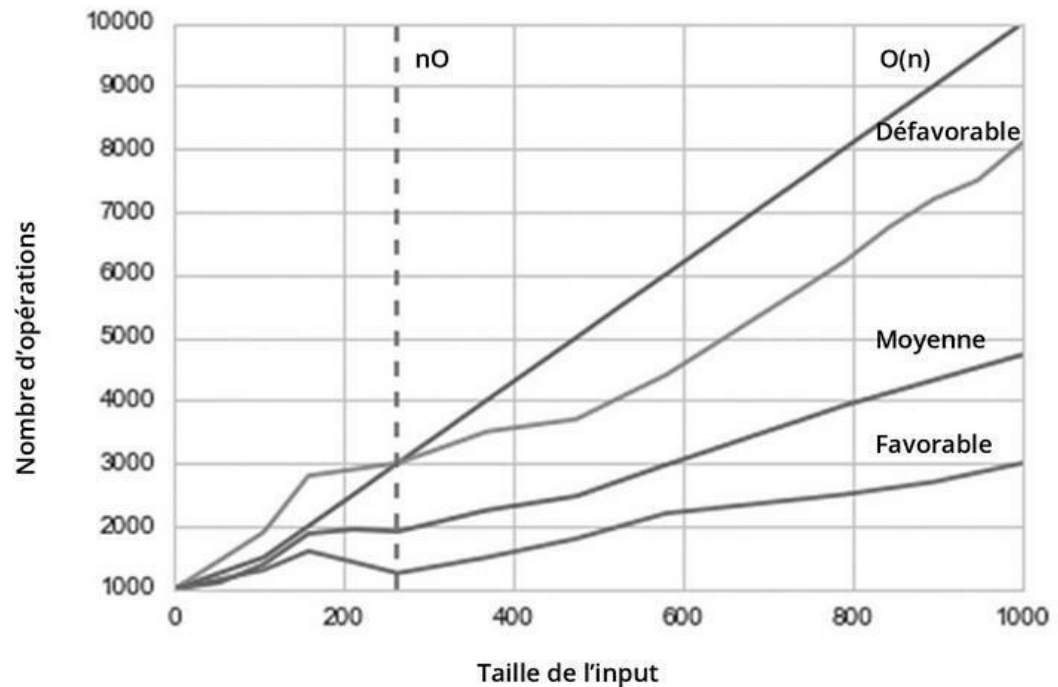


FIGURE 2-1 Complexité d'un algorithme en situation d'input favorable, moyenne et défavorable.

- » **Complexité linéaire, $O(n)$** : Le nombre d'opérations augmente avec l'input à raison d'une unité pour une unité. Un algorithme type est l'itération, consistant à faire entrer les données une seule fois et à appliquer une opération à chaque élément de ces données. Le [Chapitre 5](#) est consacré aux itérations.
- » **Complexité linéarithmique, $O(n \log n)$** : Cette complexité tient à la fois de la complexité logarithmique et de la complexité linéaire. Elle est typique de certains algorithmes intelligents utilisés pour classer des données, comme le tri fusion, le tri par tas et le tri rapide. Le [Chapitre 7](#) en présente la plus grande partie.
- » **Complexité quadratique, $O(n^2)$** : Le nombre d'opérations croît comme le carré du nombre d'inputs. Quand une itération se trouve à l'intérieur d'une autre itération (en informatique, on parle d'itérations imbriquées), la complexité est quadratique. C'est le cas, par exemple, si

vous disposez d'une liste de noms et si, afin de déterminer ceux qui se ressemblent le plus, vous comparez chaque nom à tous les autres. Certains algorithmes de classement peu efficaces présentent ce type de complexité : tri à bulles, tri par sélection, et tri par insertion. Ce niveau de complexité signifie que les algorithmes peuvent tourner pendant des heures, voire pendant des jours, avant d'aboutir à une solution.

- » **Complexité cubique, $O(n^3)$** : Le nombre d'opérations croît plus vite encore que dans le cas de la complexité quadratique, car nous avons ici de multiples itérations imbriquées. Quand un algorithme présente ce degré de complexité et quand il s'agit de traiter une quantité de données modeste (de l'ordre de 100 000 éléments), l'exécution de l'algorithme peut durer plusieurs années. Quand le nombre d'opérations est une puissance de l'input, on dit souvent que l'algorithme fonctionne en *temps polynomial*.
- » **Complexité exponentielle, $O(2^n)$** : L'algorithme effectue deux fois le nombre d'opérations antérieures pour chaque élément ajouté. Avec ce degré de complexité, même la résolution des plus petits problèmes peut prendre un temps interminable. De nombreux algorithmes de recherche exhaustive présentent une complexité exponentielle. Néanmoins, l'exemple le plus classique de ce degré de complexité est le calcul des nombres de la suite de Fibonacci (un algorithme récursif, présenté au [Chapitre 5](#)).
- » **Complexité factorielle, $O(n!)$** : Une complexité véritablement cauchemardesque, compte tenu du nombre de combinaisons possibles entre les éléments. Imaginez seulement ceci : avec un input de 100 objets et une opération que l'ordinateur exécute en 10^{-6} secondes (une vitesse raisonnable pour un ordinateur actuel), il faudrait environ 10^{140} années pour exécuter la tâche (une durée impossible, sachant que l'âge de l'univers est estimé à 10^{14}

années). Un problème de complexité factorielle bien connu est le « problème du voyageur de commerce », dans lequel un représentant doit trouver le chemin le plus court pour se rendre dans un certain nombre de villes et revenir au point de départ (présenté au [Chapitre 16](#)).

Chapitre 3

Utiliser Python pour faire de l'algorithmique

DANS CE CHAPITRE

- » Utiliser Python pour découvrir comment fonctionne un algorithme
 - » Étudier les différentes distributions de Python
 - » Installer Python sous Linux
 - » Installer Python sous OS X
 - » Installer Python sous Windows
 - » Obtenir et installer les jeux de données utilisés dans ce livre
-

Dès qu'il s'agit de s'aider de l'ordinateur pour découvrir le monde merveilleux de l'algorithmique, on dispose d'un vaste choix de langages et autres outils logiciels. Ainsi, abstraction faite de Python, de nombreux utilisateurs de l'informatique optent pour MATLAB, et un certain nombre d'autres choisissent le langage R. En fait, certains les utilisent tous les trois et comparent les résultats (pour un exemple de ce genre de comparaison, voir <https://www.r-bloggers.com/evaluating-optimization-algorithms-inmatlab-python-and-r/>). Si vous deviez choisir entre les trois, il serait cependant conseillé de bien y réfléchir et vous pourriez être bien inspiré de ne pas vous contenter d'apprendre un seul langage, mais en réalité plus de trois possibilités s'offrent à vous, et ce livre ne saurait en aucun cas les traiter toutes. En vous plongeant dans

l'algorithme, vous vous rendrez compte que vous pouvez utiliser n'importe quel langage de programmation pour écrire des algorithmes et que certains langages sont particulièrement appréciés parce qu'ils réduisent tout à des opérations simples, comme la simulation de RAM décrite au [Chapitre 2](#). Ainsi, Donald Knuth, lauréat du prix Turing, en a écrit des exemples en langage assembleur dans son ouvrage *The Art of Computer Programming* (Addison-Wesley). L'assembleur est un langage de programmation qui ressemble au langage machine, c'est-à-dire au langage qui est utilisé originairement par les ordinateurs (mais qui n'est pas compréhensible pour la plupart d'entre nous).

Si ce livre utilise Python, c'est pour plusieurs bonnes raisons, entre autres sa popularité dans le monde de l'informatique et le fait qu'il soit complet, et cependant facile à apprendre. Python est aussi un langage verbeux, plus proche de la manière dont l'utilisateur humain formule les instructions que de celle dont l'ordinateur les interprète. La première section de ce chapitre explique plus en détail le choix que nous avons fait d'utiliser Python pour les exemples, mais elle vous précise également la raison pour laquelle d'autres options sont valables et pourquoi vous pourriez avoir besoin de les retenir si vous comptez poursuivre votre immersion dans le monde de l'algorithmique.

Quand nous nous exprimons dans un langage humain, nous ajoutons des nuances de sens en employant des combinaisons de mots spécifiques compréhensibles par nos interlocuteurs. L'utilisation de telles nuances est naturelle et représente un idiome. Dans certains cas, les idiomes apparaissent aussi parce qu'un groupe tient à manifester une différence vis-à-vis d'un autre groupe. Si Noah Webster, par exemple, avait écrit et publié *A Grammatical Institute of the English Language*, c'était notamment dans le but de soustraire le public américain à l'influence de l'aristocratie britannique (pour plus de détails, voir <http://connecticuthistory.org/noah-websterand-the-dream-of-a-common-language/>). De même, les langages informatiques sont souvent proposés avec plusieurs nuances et leurs concepteurs y ajoutent à dessein des extensions qui rendent leur produit unique,

afin de donner au consommateur une raison de l'acheter, de préférence à une offre concurrente.

La deuxième section de ce chapitre vous présente diverses distributions de Python, chacune fournissant un idiome de Python. Ce livre utilise Analytics Anaconda, le produit que vous devriez utiliser pour tirer le meilleur de votre apprentissage. L'utilisation d'un autre produit, et surtout d'un autre idiome, risquerait d'engendrer des problèmes lorsque vous voudriez appliquer les exemples, comme ce qui peut se produire quand un Anglais discute avec un Américain. Un aperçu des autres distributions peut cependant vous être utile, lorsque vous voudrez accéder à des fonctionnalités qu'Anaconda ne vous offrira pas.

Les trois sections suivantes de ce chapitre vous expliquent comment installer Anaconda sur votre système. Les exemples contenus dans ce livre ont été testés sous Linux, sous Mac OS X et sous Windows. Il se peut qu'ils fonctionnent aussi dans d'autres environnements, mais comme ils n'y ont pas été testés, vous n'avez aucune assurance que ce sera le cas. En installant Anaconda conformément aux procédures exposées dans ce chapitre, vous réduisez le risque de vous retrouver avec une installation sur laquelle le code des exemples ne fonctionnerait pas. Pour pouvoir utiliser les exemples présentés dans ce livre, vous devez installer Anaconda 4.2.0 pour Python 3.5. Les autres versions d'Anaconda et de Python risquent de ne pas fonctionner avec le code de ces exemples. En effet, à l'instar des idiomes dans le cas des langages humains, ils pourraient en interpréter les instructions de façon incorrecte.

Les algorithmes traitent les données d'une façon particulière. Pour obtenir un certain résultat, il faut des données cohérentes. Heureusement, la communauté des utilisateurs de Python s'affaire à créer des jeux de données dont tout le monde peut se servir pour effectuer des essais. C'est ce qui permet aux utilisateurs de reproduire des résultats obtenus par d'autres sans être obligés de télécharger des jeux de données personnalisés à partir d'une source inconnue. La dernière section de ce chapitre vous assiste

dans l'obtention et l'installation des jeux de données nécessaires pour utiliser ces exemples.

Prendre la mesure des avantages de Python

Pour pouvoir faire fonctionner des algorithmes sur un ordinateur, il vous faut un moyen de communiquer avec la machine. Si nous étions des personnages de *Star Trek*, il vous suffirait sans doute de dire à l'ordinateur ce que vous désirez, et il effectuerait la tâche pour vous avec diligence. D'ailleurs, dans *Star Trek IV*, Scotty semble dérouté par l'absence d'interface vocale avec l'ordinateur (voir <http://www.davidalison.com/2008/07/keyboard-vs-mouse.html>). Le fait est qu'aujourd'hui vous avez encore besoin de vous servir de la souris et du clavier, ainsi que d'un langage spécial, pour pouvoir communiquer vos idées à l'ordinateur, sachant que celui-ci n'est pas près de faire un effort pour communiquer avec vous. Python est un langage qui facilite la tâche aux utilisateurs qui ne sont pas des développeurs, mais il y en a d'autres. Les paragraphes qui suivent vous expliquent pourquoi ce livre utilise Python et quelles seraient pour vous les autres possibilités.

Comprendre pourquoi ce livre utilise Python

Tous les langages informatiques actuels traduisent les algorithmes sous une forme que la machine est capable de traiter. C'est particulièrement évident quand on songe à des langages comme Algol (ALGOritmic Language) et le Fortran (FORmula TRANslation). Souvenons-nous de la définition de l'algorithme au [Chapitre 1](#), une succession d'étapes destinée à résoudre un problème. La méthode utilisée pour réaliser cette traduction diffère d'un langage à un autre, et les techniques utilisées par

certains langages sont assez difficiles à comprendre, la moindre tentative exigeant une connaissance spécialisée.



Les ordinateurs ne parlent qu'un langage, le *langage machine* (constitué de 0 et de 1, et que la machine interprète pour exécuter des tâches), et ce langage est si inaccessible à l'être humain que les premiers développeurs se sont empressés de créer un vaste ensemble d'alternatives. Les langages informatiques servent à faciliter la communication des humains avec les ordinateurs. Par conséquent, s'il vous arrive d'échouer à faire fonctionner quelque chose, peut-être est-ce un problème de langage. Il vaut toujours mieux être en mesure d'utiliser plus d'un langage, afin de pouvoir effectuer cette communication sans difficulté. Il se trouve que Python est un des langages les plus adaptés pour les utilisateurs qui travaillent dans des disciplines autres que le développement d'applications informatiques.

Python est la vision d'une seule personne, Guido van Rossum (voir sa page d'accueil à l'adresse <https://gvanrossum.github.io/>). Vous serez peut-être surpris d'apprendre que Python existe depuis un certain temps déjà : Guido s'est lancé dans son développement en décembre 1989 en vue de remplacer le langage ABC. On dispose de peu d'informations concernant les objectifs précis qui ont motivé le développement de Python, qui permet de créer des applications comme le permettait ABC, mais en utilisant moins de code. Néanmoins, les possibilités de Python dépassent de loin celles d'ABC en matière de développement d'applications de tous types, et ce qui le distingue d'ABC, c'est aussi la possibilité d'utiliser quatre styles de programmation différents. Pour faire bref, Guido a pris ABC comme point de départ, et le trouvant trop limité, il a créé un nouveau langage qui ne présente plus ces limitations. C'est là l'exemple d'un nouveau langage vraiment meilleur que ceux qui l'ont précédé.

Python a franchi un certain nombre d'itérations et suit actuellement deux parcours de développement. La série 2.x est compatible avec les versions précédentes de Python, pas la série 3.x. La question de la compatibilité est liée à la manière dont on utilise Python pour exécuter des tâches algorithmiques, sachant

que certains modules ne pourront pas fonctionner avec la série 3.x. En outre, certaines versions utilisent des licences différentes, sachant que Guido a travaillé dans plusieurs sociétés au cours du développement de Python. Vous pouvez consulter une liste des versions, avec leurs licences respectives, à l'adresse <https://docs.python.org/3/license.html>. La Python Software Foundation (PSF) détient les licences de toutes les versions actuelles de Python. Par conséquent, à moins que vous n'utilisiez une version plus ancienne, vous n'avez pas lieu de vous soucier du problème de la licence.



Guido a commencé à concevoir Python dans le cadre d'un *projet skunkworks* (un projet développé par un petit groupe de personnes peu structuré). L'idée était de créer Python aussi rapidement que possible, mais d'en faire un langage flexible, pouvant fonctionner dans n'importe quel environnement et qui présenterait un potentiel d'extension important. Python possède en effet ces propriétés, et bien d'autres encore. Bien sûr, il reste toujours des difficultés, comme savoir quelle partie exacte du système sous-jacent doit être exposée. Pour en savoir plus sur la philosophie ayant présidé à la conception de Python, consultez la page <http://python-history.blogspot.com/2009/01/pythonsdesign-philosophy.html>. L'historique de Python, à l'adresse <http://python-history.blogspot.com/2009/01/introduction-and-overview.html>, comporte aussi des informations utiles.

Les objectifs de développement (ou de conception) initiaux, concernant Python, ne correspondent pas tout à fait à ce qui s'est passé depuis. À l'origine, Guido avait envisagé Python comme un deuxième langage pour les développeurs qui avaient besoin de produire du code à usage unique, mais qui pouvaient difficilement atteindre leurs objectifs en utilisant un langage de script. Le public cible initial était constitué des gens qui programmaient en langage C. À propos des orientations initiales, vous pouvez lire une interview à l'adresse <http://www.artima.com/intv/pyscale.html>.

Aujourd'hui, on trouve un certain nombre d'applications écrites en Python, aussi l'idée de l'utiliser uniquement pour le script ne s'est pas concrétisée. Vous pouvez même trouver des listes

d'applications à l'adresse <https://www.python.org/about/apps/> ainsi qu'à l'adresse <https://www.python.org/about/success/>.

Naturellement, compte tenu de tous ces exemples, les utilisateurs adoptent Python avec enthousiasme. Vous trouverez des listes de propositions d'amélioration (Python Enhancement Proposals, ou PEP) à l'adresse <http://legacy.python.org/dev/peps/>. On ne peut pas dire lesquelles de ces propositions auront une suite, mais tout cela prouve que Python est un langage vivant et en pleine croissance, qui va continuer à apporter des fonctionnalités dont les développeurs ont véritablement besoin pour créer toutes sortes de belles applications.

Travailler avec MATLAB

Python présente un certain nombre d'avantages par rapport à beaucoup d'autres langages. Il permet d'utiliser différents styles de codage, sa flexibilité est remarquable, il est très évolutif, mais il reste un langage de programmation. Si vous ne voulez vraiment pas utiliser un langage de programmation, il y a d'autres possibilités comme MATLAB (<https://www.mathworks.com/products/matlab/>), qui est plus axé sur les algorithmes. MATLAB reste une variété de langage de script, et pour pouvoir lui faire exécuter des tâches importantes, vous avez tout de même besoin de maîtriser quelques notions de codage, pas autant cependant qu'avec Python.

Un des plus grands problèmes que pose l'utilisation de MATLAB, c'est le prix à payer. Contrairement à Python, MATLAB exige de votre part un investissement monétaire (à propos du prix de la licence, voir <https://www.mathworks.com/pricing-licensing/>). L'environnement de développement est nettement plus facile à maîtriser, mais là comme ailleurs, rien n'est gratuit et vous devez tenir compte de la différence de coût pour déterminer le produit que vous utiliserez.

MATLAB suscite beaucoup d'intérêt, compte tenu de ses points forts et de ses points faibles par rapport à Python. Il n'y aurait pas

la place dans ce livre de présenter une comparaison complète, mais vous trouverez un bon aperçu à l'adresse http://www.pyzo.org/python_vs_matlab.html. Par ailleurs, vous pouvez appeler des modules Python depuis MATLAB en utilisant les techniques présentées sur la page <https://www.mathworks.com/help/matlab/call-pythonlibraries.html>. MATLAB est d'ailleurs compatible avec tout ce qui suit :

- » MEX (<https://www.mathworks.com/help/matlab/call-mex-filefunctions.html>)
- » C (<https://www.mathworks.com/help/matlab/using-c-shared-libraryfunctions-in-matlab-.html>)
- » Java (<https://www.mathworks.com/help/matlab/using-java-librariesin-matlab.html>)
- » NET (<https://www.mathworks.com/help/matlab/using-net-librariesin-matlab.html>)
- » COM (<https://www.mathworks.com/help/matlab/using-com-objectsin-matlab.html>) \$

Vous n'êtes donc pas obligé de faire un choix entre MATLAB et Python (ou un autre langage), mais plus vous utiliserez les fonctionnalités de Python, plus il vous deviendra facile de travailler simplement avec Python et non plus avec MATLAB. Pour en savoir plus sur MATLAB, consultez l'ouvrage *MATLAB For Dummies*, de Jim Sizemore et John Paul Mueller (Wiley).

Étudier d'autres environnements de test d'algorithmes

Une troisième possibilité intéressante pour travailler avec les algorithmes est le langage de programmation R, qui comme Python, est gratuit. Ce langage aussi est compatible avec un grand nombre de modules et présente une grande flexibilité. Certaines

constructions de programmation sont cependant différentes, et R est parfois considéré comme plus difficile à utiliser que Python. Aux yeux de la plupart des utilisateurs, c'est R qui l'emporte pour les calculs statistiques, mais le caractère polyvalent de Python présente des atouts majeurs (voir les articles des pages <https://www.datacamp.com/community/tutorials/r-or-python-for-data-analysis> et <http://www.kdnuggets.com/2015/05/r-vs-python-data-science.html>). Le plus fort soutien dont bénéficie Python auprès des informaticiens est aussi un grand avantage.

Comme mentionné précédemment, vous pouvez utiliser n'importe quel langage de programmation pour réaliser des travaux liés à l'algorithmique, mais la plupart des langages sont conçus avec une orientation particulière. Ainsi, par exemple, vous pouvez effectuer ces travaux en utilisant un langage comme SQL (*Structured Query Language*), toutefois, sachant que ce langage est axé sur la gestion de données, il se peut que certaines tâches d'algorithmique deviennent alambiquées et difficiles à mener à bien. Ce qui fait défaut à SQL, c'est notamment la capacité de recueillir aisément les données et d'effectuer certaines des traductions et transformations que nécessitent des travaux spécifiquement basés sur l'algorithmique. En un mot, vous devez choisir un langage en fonction de ce que vous comptez faire. Ce livre utilise Python parce qu'il s'agit véritablement du meilleur langage général pour réaliser les tâches en question, mais il est important que vous vous rendiez compte qu'à un moment donné, vous aurez peut-être besoin d'utiliser un autre langage.

Découvrir les modules de Python

Vous pouvez très probablement obtenir une copie générique de Python et y ajouter tous les modules nécessaires pour travailler avec des algorithmes. Le processus peut être difficile car vous devez vous assurer de vous procurer les bonnes versions de tous ces modules pour une garantie de réussite. Vous devrez aussi mettre en place la configuration requise pour que tous les modules soient accessibles au moment où vous en aurez besoin.

Heureusement, tout cela n'est pas nécessaire car de nombreux produits Python qui conviennent bien pour l'algorithmique sont à votre disposition. Avec ces produits, vous aurez tout le nécessaire pour vous lancer dans vos projets d'algorithmique.



Vous pouvez utiliser n'importe lequel des modules mentionnés dans les sections suivantes pour travailler sur les exemples de ce livre. Cependant, le code source du livre et le code source téléchargeable proviennent de Continuum Analytics Anaconda 4.2.0, car ce module fonctionne dans les différents environnements pris en compte dans ce livre : Linux, Mac OS X, et Windows. Le livre ne mentionne pas un module particulier dans les chapitres qui suivent, mais les captures d'écran montrent ce que l'on obtient en utilisant Anaconda sous Windows. Vous devrez peut-être adapter le code si vous utilisez un autre module, et ce qui apparaîtra à l'écran sera différent si vous utilisez Anaconda avec un autre système d'exploitation.



Windows 10 présente de sérieux problèmes d'installation quand on travaille avec Python. Je les mentionne sur mon blog (c'est John Mueller qui parle), sur la page <http://blog.johnmuellerbooks.com/2015/10/30/python-and-windows-10/>. Étant donné qu'un si grand nombre de lecteurs de mes autres ouvrages sur Python m'ont fait part de leurs impressions en disant que Windows 10 ne constituait pas un bon environnement, je ne peux pas recommander Windows 10 comme plateforme pour Python dans le cadre de ce livre. Si vous utilisez Windows 10, sachez simplement que pour installer Python, vous n'avez pas fini de rencontrer des problèmes.

Obtenir Anaconda

La version de base de la suite Anaconda peut être téléchargée gratuitement depuis la page <https://www.continuum.io/downloads>. Cliquez tout simplement sur « Download Anaconda ». Vous devez fournir une adresse électronique pour obtenir votre copie d'Anaconda, après quoi vous arrivez sur une autre page, sur

laquelle vous pouvez sélectionner votre système d'exploitation et le programme d'installation pour ce système. Anaconda prend en charge les systèmes suivants :

- » Windows 32 bits et 64 bits (le programme d'installation ne vous proposera peut-être qu'une des deux versions, selon la version de Windows qu'il aura détectée) ;
- » Linux 32 bits et 64 bits ;
- » Mac OS X 64 bits.

Le support de Python 3.5 étant devenu meilleur que les versions 3.x, Python 3.x et 2.x sont tous les deux présents sur le site d'Analytics. Ce livre utilise Python 3.5 parce que le support est aujourd'hui assez substantiel et assez stable pour supporter tous les exemples de programmation, et parce que Python 3.x représente l'avenir de Python.



Vous pouvez obtenir Anaconda avec des versions plus anciennes de Python. Si vous souhaitez utiliser une version de Python plus ancienne, cliquez sur le lien de l'archive du programme d'installation, vers le bas de la page. N'utilisez une version plus ancienne de Python qu'en cas de nécessité.

Le programme d'installation Miniconda peut certes vous faire économiser du temps en limitant le nombre de fonctionnalités à installer. Cependant, en cherchant à n'installer que les modules dont vous aurez vraiment besoin, vous risqueriez de vous tromper et de perdre du temps. De manière générale, mieux vaut procéder à une installation complète afin d'être sûr d'avoir tout ce qu'il vous faut pour vos projets. Sur la plupart des systèmes, même une installation complète ne sera pas longue ni difficile.

La version gratuite est amplement suffisante pour travailler avec ce livre. Néanmoins, sur le site, un certain nombre de compléments sont disponibles, qui peuvent vous permettre de créer de belles applications. Si vous ajoutez le module Accelerate, par exemple, vous pourrez effectuer des traitements multiconducteurs et GPU. Le site d'Anaconda fournit des détails

concernant l'utilisation de ces modules complémentaires, qui n'entre pas dans le cadre de ce livre.

Si Enthought Canopy Express vous intéresse...

Enthought Canopy Express est un produit gratuit destiné au développement d'applications techniques et scientifiques à l'aide de Python. Vous pouvez l'obtenir à l'adresse <https://www.enthought.com/canopy-express/>. Cliquez sur « Download Free », sur la page d'accueil, et vous verrez apparaître une liste des versions téléchargeables. Seul Canopy Express est gratuit : la version intégrale de Canopy est payante. Vous pouvez utiliser Canopy Express pour travailler sur les exemples de ce livre. Canopy Express fonctionne sur les plateformes suivantes :

- » Windows 32 bits et 64 bits ;
- » Linux 32 bits et 64 bits ;
- » Mac OS X 32 bits et 64 bits.

Choisissez la plateforme et la version que vous voulez télécharger. Quand vous cliquez sur « Download Canopy Express », un formulaire facultatif apparaît, dans lequel vous pouvez saisir des renseignements vous concernant. Le téléchargement commence automatiquement, même si vous n'avez rien saisi dans le formulaire.

Un des avantages de Canopy Express est qu'Enthought s'attache à fournir un soutien aux étudiants et aux enseignants. Il est possible aussi de prendre des cours, y compris des cours en ligne, afin d'apprendre à utiliser Canopy Express de différentes manières (voir <https://training.enthought.com/courses>).

... Ou bien Python(x,y)

L'environnement de développement intégré Python(x,y) est un projet communautaire hébergé par Google à l'adresse <http://python-xy.github.io/>. C'est un produit pour Windows uniquement. Par conséquent, vous ne pourrez pas facilement l'utiliser pour des besoins multiplateformes (en fait, il est compatible uniquement avec Windows Vista, Windows 7 et Windows 8). Il s'accompagne cependant d'une série complète de modules, et vous pouvez facilement l'utiliser dans le cadre de ce livre si le cœur vous en dit.

Sachant que Python(x,y) fonctionne sous la licence publique générale GNU (GPL) v3 (voir <http://www.gnu.org/licenses/gpl.html>), vous n'avez pas à vous soucier des modules complémentaires, de la formation ni d'autres fonctionnalités payantes. Personne ne viendra sonner à votre porte pour tenter de vous vendre quelque chose. En outre, vous avez accès à tous les codes sources de Python(x,y), ce qui vous permet d'effectuer des modifications si vous le désirez.

Il y a aussi WinPython

Comme son nom l'indique, WinPython est un produit pour Windows uniquement. Vous pouvez le télécharger à l'adresse suivante : <http://winpython.sourceforge.net/>. Ce produit est en fait une émanation de Python(x,y) et il n'est pas destiné à le remplacer. Bien au contraire, WinPython est simplement un moyen plus flexible de travailler avec Python(x,y). Pour en savoir davantage sur ce qui a motivé sa création, consultez la page <http://sourceforge.net/p/winpython/wiki/Roadmap/>.

Ce qu'il faut retenir concernant ce produit, c'est le gain en flexibilité aux dépens de la convivialité et de l'intégration sur les plateformes. Néanmoins, pour les développeurs qui ont besoin de conserver plusieurs versions d'un environnement de développement intégré, WinPython peut faire une différence significative. Si vous utilisez WinPython avec ce livre, accordez une attention particulière aux problèmes de configuration, faute de

quoi, même le code téléchargeable aurait peu de chances de fonctionner.

Installer Python sous Linux

Pour installer Anaconda sous Linux, utilisez la ligne de commande : en effet, il n'y a aucune possibilité d'installation graphique. Avant de procéder à l'installation, vous devez télécharger une copie du logiciel Linux à partir du site de Continuum Analytics. Vous trouverez les informations nécessaires dans la section « Obtenir Anaconda », précédemment dans ce chapitre. La procédure suivante devrait bien fonctionner sur n'importe quel système Linux, que vous utilisiez la version 32 bits ou la version 64 bits d'Anaconda :

- 1. Ouvrez une copie de Terminal.**

La fenêtre de Terminal apparaît.

- 2. Changez la destination de la copie d'Anaconda qui sera téléchargée sur votre système.**

Le nom du fichier varie, mais c'est normalement [Anaconda3-4.2.0-Linux-x86.sh](#) pour les systèmes 32 bits et [Anaconda3-4.2.0-Linux-x86_64.sh](#) pour les systèmes 64 bits. Le numéro de version est inscrit dans le nom de fichier. En l'occurrence, le nom du fichier fait référence à la version 4.2.0, qui est la version utilisée pour ce livre. Si vous utilisez une autre version, vous risquez de rencontrer des problèmes avec le code source et quelques ajustements seront nécessaires.

- 3. Sélectionnez « bash [Anaconda3-4.2.0-Linux-x86.sh](#) »**
(pour la version

32 bits) ou « bash [Anaconda3-4.2.0-Linux-x86_64.sh](#) »
(pour la version

64 bits) et appuyez sur la touche Entrée. L'assistant d'installation vous demande d'accepter les conditions de

licence pour l'utilisation d'Anaconda.

- 4. Lisez les conditions et acceptez-les en utilisant la méthode requise pour votre version de Linux.**

L'assistant d'installation vous demande de préciser le chemin d'installation pour Anaconda. Dans ce livre, on suppose que vous avez choisi le chemin par défaut, ~/anaconda. Si vous choisissez un autre emplacement, vous devrez modifier certaines procédures par la suite afin de les adapter à votre configuration.

- 5. Précisez (le cas échéant) l'emplacement d'installation et appuyez sur la touche Entrée (ou cliquez sur « Next »).**

Le processus d'extraction de l'application commence. Attendez le message indiquant que l'extraction est terminée.

- 6. Ajoutez le chemin d'installation dans la spécification du chemin (PATH statement) en utilisant la méthode requise pour votre version de Linux.**

Vous pouvez commencer à utiliser Anaconda.

Installer Python sous macOS **(<https://fr.wikipedia.org/wiki/MacOS>)**

L'installation sous Mac OS X se fait selon un seul format : 64 bits. Avant de commencer l'installation, vous devez télécharger une copie du logiciel pour Mac à partir du site de Continuum Analytics. Vous trouverez les informations nécessaires dans la section « Obtenir Anaconda », précédemment dans ce chapitre.

Les fichiers d'installation peuvent se présenter sous deux formes, selon que l'on utilise le programme d'installation graphique ou la ligne de commande. La version de la ligne de commande fonctionne de façon similaire à la version mentionnée dans la section « Installer Python sous Linux ». Pour installer

Anaconda 64 bits sur un système Mac à l'aide du programme d'installation graphique, procédez comme suit :

1. **Trouvez la copie téléchargée d'Anaconda dans votre système.**

Le nom du fichier varie, mais il apparaît normalement sous la forme suivante : Anaconda3-4.2.0-MacOSX-x86_64.pkg. Le numéro de version est inscrit dans le nom du fichier. En l'occurrence, le nom du fichier fait référence à la version 4.2.0, qui est la version utilisée pour ce livre. Si vous utilisez une autre version, vous risquez de rencontrer des problèmes avec le code source et quelques ajustements seront nécessaires.

2. **Faites un double-clic sur le fichier d'installation.**

Une boîte de dialogue apparaît.

3. **Cliquez sur « Continue ».** L'assistant d'installation vous demande si vous désirez lire le fichier « Read Me ». Vous pourrez le lire plus tard. Pour l'instant, vous pouvez passer cette étape sans risque.

4. **Cliquez sur « Continue ».**

L'assistant affiche un accord de licence. Ne manquez pas de le parcourir afin de connaître les conditions d'utilisation.

5. **Cliquez sur « I Agree » si vous acceptez l'accord de licence.**

L'assistant vous demande de préciser une destination pour l'installation. Le programme effectue un contrôle pour savoir si l'installation est faite pour un utilisateur unique ou pour un groupe d'utilisateurs.

Il se peut qu'apparaisse un message d'erreur signalant que vous ne pouvez pas installer Anaconda sur votre système. Cela peut se produire en raison d'un bogue dans le programme d'installation, et cela n'a rien à voir avec votre système. Pour vous débarrasser du message d'erreur,

choisissez l'option « Install Only for Me ». Sur un système Mac, il n'est pas possible d'installer Anaconda pour un groupe d'utilisateurs.

6. Cliquez sur « Continue ».

Le programme d'installation affiche une boîte de dialogue comportant des options pour changer le type d'installation. Cliquez sur « Change Install Location » si vous voulez modifier le chemin d'installation d'Anaconda (dans ce livre, on suppose que vous optez pour le chemin par défaut, `~/anaconda`). Cliquez sur « Customize » si vous voulez personnaliser l'installation. Vous pourriez choisir, par exemple, de ne pas ajouter Anaconda à la spécification du chemin (*PATH statement*). Cependant, dans ce livre, on suppose que vous avez choisi l'installation par défaut et il n'existe aucune bonne raison de modifier les options de l'installation, sauf si vous avez déjà installé ailleurs une autre copie de Python 3.5.

7. Cliquez sur « Install ».

L'installation commence. Une barre vous indique sa progression. Attendez qu'une boîte de dialogue vous signale que l'installation est terminée.

8. Cliquez sur « Continue ».

Vous pouvez commencer à utiliser Anaconda.

Installer Python sous Windows

Anaconda s'installe au moyen d'une application graphique pour Windows, donc une bonne installation suppose l'utilisation d'un assistant, comme pour toute autre installation. Naturellement, il vous faut une copie du fichier d'installation avant de commencer, et vous trouverez les informations nécessaires à l'installation dans la section « Obtenir Anaconda », précédemment dans ce chapitre. La procédure suivante devrait fonctionner parfaitement sous

Windows, que vous utilisiez la version 32 bits ou la version 64 bits d'Anaconda :

- 1. Trouvez la copie d'Anaconda téléchargée sur votre système.**

Le nom du fichier varie, mais il apparaît normalement sous la forme suivante : Anaconda3-4.2.0- Windows-x86.exe pour les systèmes 32 bits et Anaconda3-4.2.0-Windows-x86_64.exe pour les systèmes 64 bits. Le numéro de version est inscrit dans le nom du fichier. En l'occurrence, le nom du fichier fait référence à la version 4.2.0, qui est la version utilisée pour ce livre. Si vous utilisez une autre version, vous risquez de rencontrer des problèmes avec le code source et quelques ajustements seront nécessaires.

- 2. Faites un double-clic sur le fichier d'installation.**

(Si vous voyez apparaître une boîte de dialogue d'alerte de sécurité, confirmez que vous voulez l'ouverture du fichier.) Une boîte de dialogue d'installation d'Anaconda 4.2.0 va alors s'ouvrir, similaire à celle de la [Figure 3-1](#). La boîte de dialogue que vous allez précisément voir apparaître dépendra de la version du programme d'installation qui aura été téléchargée. Avec un système d'exploitation 64 bits, il vaut toujours mieux utiliser la version 64 bits d'Anaconda afin d'obtenir les meilleures performances. S'il s'agit de la version

[64](#) bits du produit, la première boîte de dialogue vous l'indique.

- 3. Cliquez sur « Next ».**

L'assistant d'installation affiche un accord de licence. Ne manquez pas de le parcourir afin de connaître les conditions d'utilisation.

- 4. Cliquez sur « I Agree » si vous acceptez l'accord de licence.**

On vous demande alors quel type d'installation vous préférez ([Figure 3-2](#)). Dans la plupart de cas, il s'agit d'installer le produit pour votre usage exclusif. Il y a exception si votre système est utilisé par plusieurs personnes qui ont toutes besoin d'avoir accès à Anaconda.

5. **Choisissez un type d'installation puis cliquez sur « Next ».** L'assistant vous demande où installer Anaconda sur le disque ([Figure 3-3](#)). Dans ce livre, on suppose que vous allez utiliser le chemin par défaut. Si vous choisissez un autre emplacement, vous devrez modifier certaines procédures par la suite afin de les adapter à votre configuration.



FIGURE 3-1 Le processus d'installation vous indique si vous avez la version 64 bits.

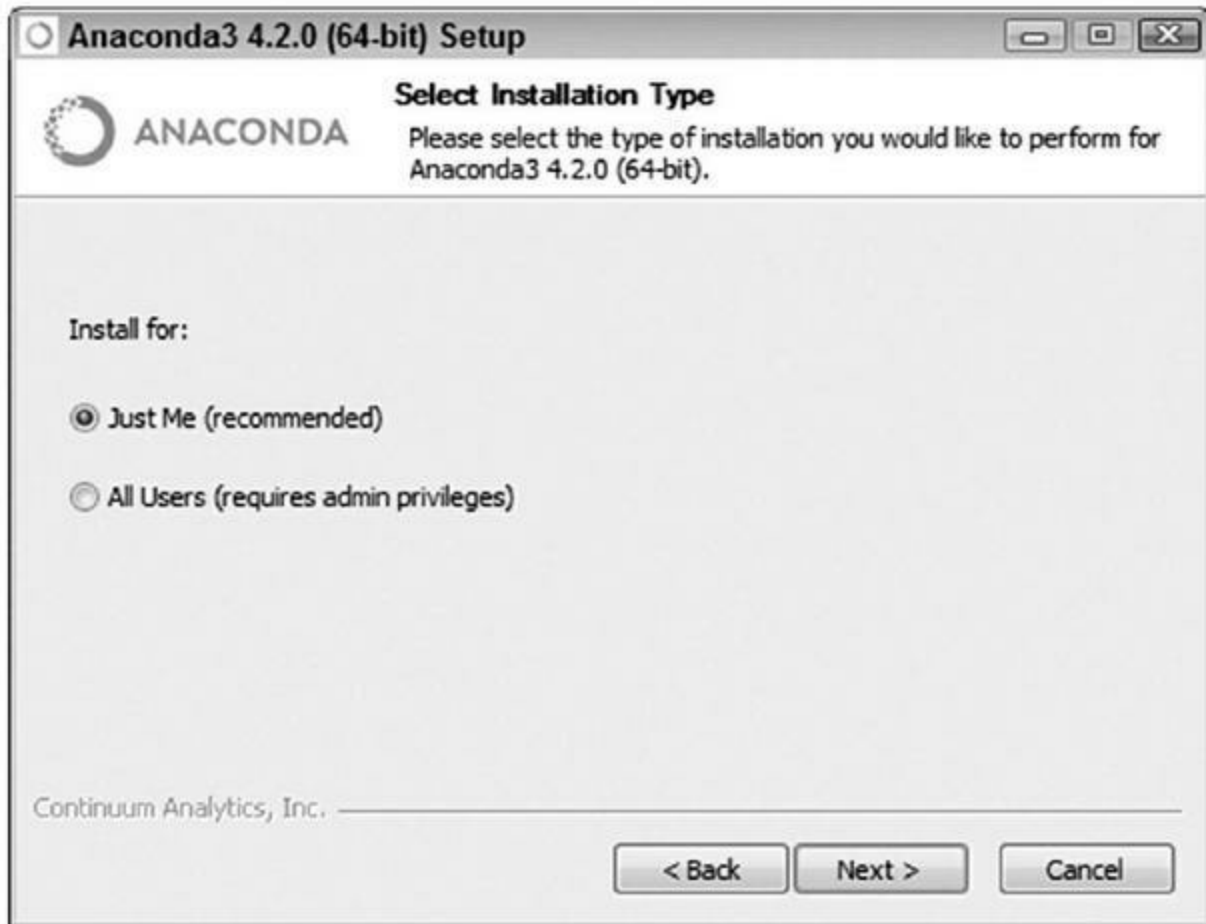


FIGURE 3-2 Dites à l'assistant comment installer Anaconda sur votre système.

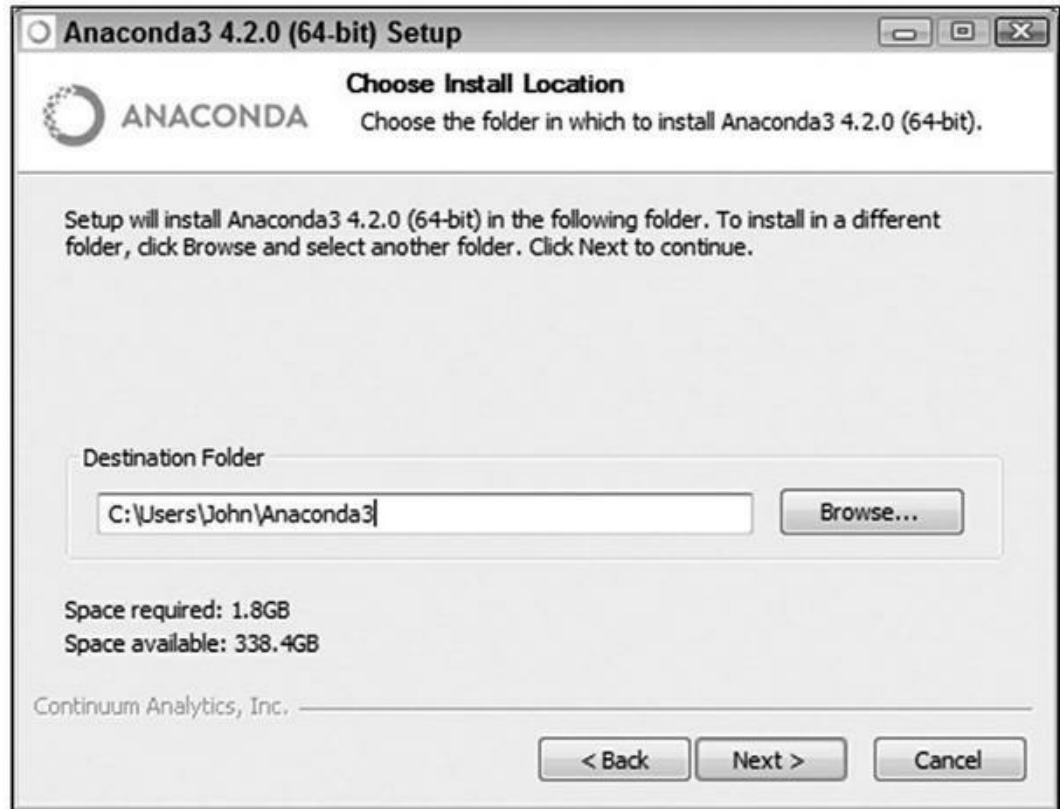


FIGURE 3-3 Spécifiez un emplacement pour l’installation.

6. Précisez (le cas échéant) l’emplacement d’installation et cliquez sur « Next ».

Les options d’installation avancée apparaissent ([Figure 3-4](#)). Ces options sont sélectionnées par défaut, et dans la plupart des cas, il n’existe aucune bonne raison de les changer. Vous pourriez avoir besoin de les changer si Anaconda n’assurait pas l’installation par défaut de Python 3.5 (ou de Python 2.7). Cependant, dans ce livre, on suppose que vous avez installé Anaconda en utilisant les options par défaut.

7. Modifiez les options d’installation avancée (le cas échéant) puis cliquez sur le bouton Install.

Vous voyez apparaître une boîte de dialogue d’installation avec une barre de progression. L’installation peut durer quelques minutes. Préparez-vous une tasse de café et lisez

une bande dessinée en attendant. Dès que l'installation sera terminée, le bouton Next sera activé.

8. **Cliquez sur le bouton Next.** L'assistant vous prévient que l'installation a été réalisée avec succès.

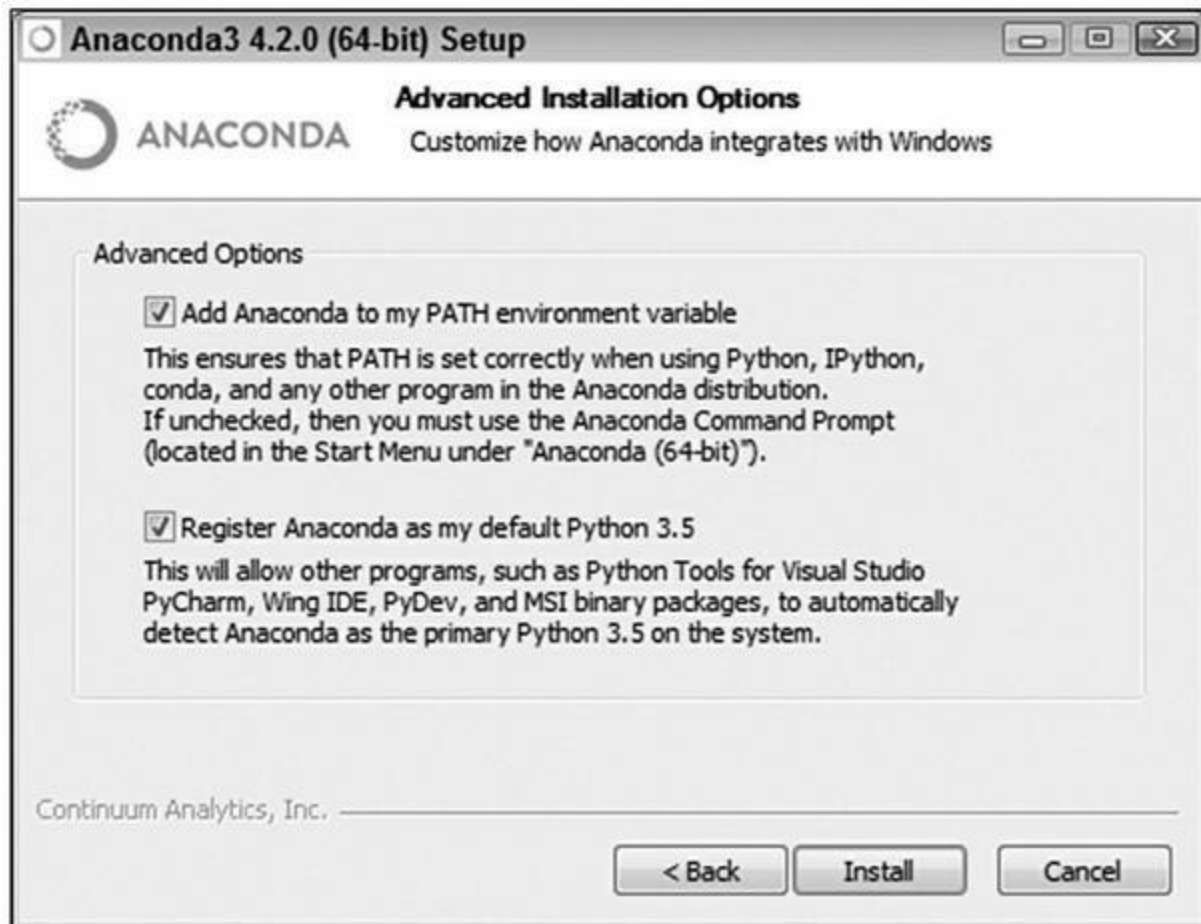


FIGURE 3-4 Configurez les options d'installation avancée.

UN MOT SUR LES CAPTURES D'ÉCRAN

Au fil de votre progression dans ces travaux, vous allez utiliser l'environnement de développement intégré (IDE) de votre choix pour ouvrir les fichiers Python et Jupyter Notebook contenant le code source du livre. Chaque capture d'écran comportant des informations spécifiques à cet IDE est liée à Anaconda, car Anaconda tourne sur les

trois plateformes concernées par ce livre. Si c'est Anaconda qui est utilisé, cela ne signifie pas nécessairement qu'il s'agit du meilleur IDE ni que les auteurs le recommandent de quelque manière que ce soit : simplement, Anaconda est un produit de démonstration performant.

Quand on travaille avec Anaconda, le nom de l'environnement graphique, Jupyter Notebook, est le même précisément sur les trois plateformes, et vous ne remarquerez même pas de différence significative dans la présentation (Jupyter Notebook étant une évolution de IPython, les ressources en ligne pourront faire référence à IPython Notebook). Les différences que vous pourrez observer sont mineures, et vous devrez les ignorer tout au long de l'utilisation de ce livre. À cet égard, ce livre utilise surtout des captures d'écran Windows 7. Si vous utilisez Linux, Mac OS X ou une autre version de Windows, vous pouvez vous attendre à quelques différences au niveau de la présentation, mais ces différences ne devraient pas vous poser de problème lorsque vous travaillerez avec les exemples.

9. Cliquez sur le bouton Finish.

Vous pouvez commencer à utiliser Anaconda.

Télécharger les jeux de données et le code exemple

Ce livre concerne l'utilisation de Python pour des tâches d'apprentissage machine. Bien sûr, vous pouvez consacrer tout votre temps à créer le code exemple *ex nihilo* et à le déboguer, pour découvrir ensuite seulement son lien avec l'apprentissage machine, mais vous pouvez aussi opter pour la facilité et télécharger le code déjà écrit à partir du site américain des Nuls (pour plus de détails, consulter l'introduction) afin de pouvoir vous mettre directement au travail. De même, créer des jeux de données assez fournis pour les besoins de l'apprentissage de l'algorithmique demanderait beaucoup de temps. Heureusement, vous pouvez facilement disposer de jeux de données préétablis et

normalisés, grâce à des fonctionnalités présentes dans certains des modules de science des données (et cela convient très bien également pour toutes sortes d'applications, y compris pour l'apprentissage de l'algorithmique). Les sections qui suivent vous assistent dans le téléchargement et l'utilisation du code exemple et des jeux de données, ce qui vous permet de gagner du temps et de vous lancer directement dans l'étude des tâches liées aux algorithmes.

Utiliser Jupyter Notebook

Pour faciliter la mise en œuvre du code relativement complexe qui est dans ce livre, vous allez utiliser Jupyter Notebook. Cette interface vous permet de créer facilement des fichiers Python pouvant contenir un nombre quelconque d'exemples, qui peuvent être utilisés chacun indépendamment des autres. Le programme tourne dans votre navigateur, si bien que la plateforme utilisée pour le développement n'a pas d'importance : tant qu'elle comporte un navigateur, ce ne devrait pas être un souci.

Lancer Jupyter Notebook

La plupart des plateformes affichent une icône sur laquelle il suffit de cliquer pour accéder à Jupyter Notebook. Sous Windows, par exemple, sélectionnez Démarrer → Tous les programmes → Anaconda 3 → Jupyter Notebook. La [Figure 3-5](#) représente l'interface quand elle est visualisée dans le navigateur Firefox. Son apparence précise sur votre écran dépend du navigateur que vous utilisez et du type de plateforme que vous avez installé.

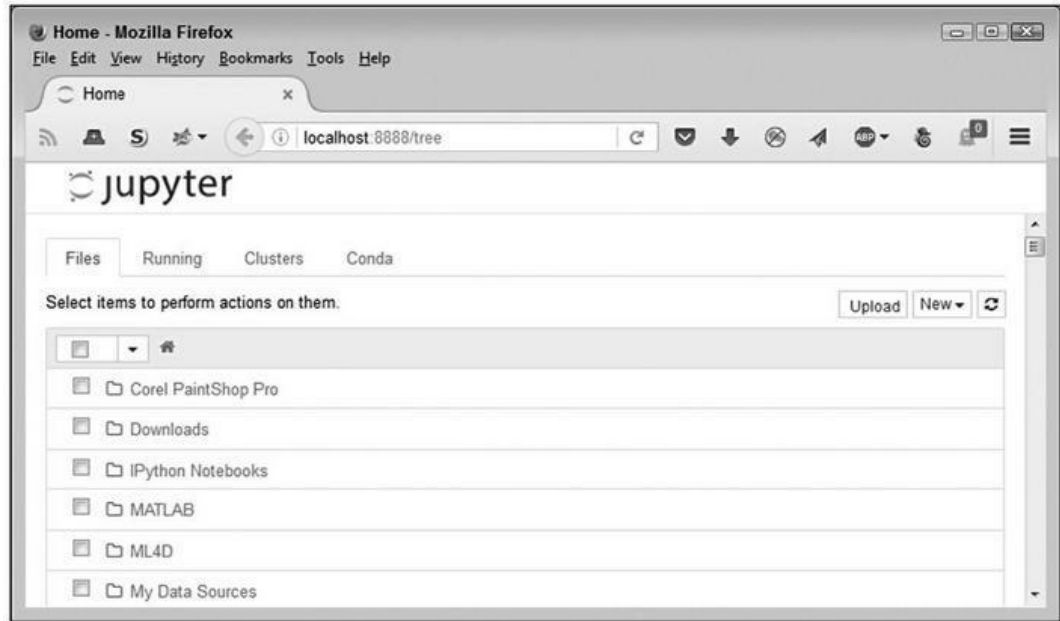


FIGURE 3-5 Jupyter Notebook présente une méthode facile pour créer des exemples d'apprentissage machine.

Si votre configuration ne vous permet pas un accès facile par une icône, vous pouvez procéder comme suit pour accéder à Jupyter Notebook :

1. **Dans votre système, ouvrez une invite de commande ou une fenêtre de terminal.**

Une fenêtre apparaît, dans laquelle vous pouvez écrire une commande.

2. **Changez de répertoire pour accéder au répertoire \Anaconda3\Scripts.**

Avec la plupart des systèmes, vous pouvez vous servir de la commande CD pour cela.

3. **Tapez `python jupyter-notebook-script.py` et appuyez sur la touche Entrée.**

La page Jupyter Notebook s'ouvre dans votre navigateur.

Arrêter le serveur Jupyter Notebook

Quelle que soit la façon dont vous lancez Jupyter Notebook (ou simplement Notebook, comme on le désignera dans le reste de ce livre), le système ouvre généralement une invite de commande ou une fenêtre de terminal pour le faire apparaître. Dans cette fenêtre se trouve un serveur qui fait fonctionner l'application. Après avoir fermé la fenêtre du navigateur une fois la session terminée, activez la fenêtre du serveur et pressez Ctrl + C ou Ctrl + Break pour arrêter le serveur.

Définir le répertoire du code

Le code que vous créez et utilisez dans le cadre de ce livre sera stocké dans un répertoire sur votre disque dur, un peu comme on range un document dans une armoire de classement. Notebook ouvre un tiroir, en sort un dossier, et vous présente le code. Vous pouvez le modifier, essayer tel ou tel exemple dans le dossier, y ajouter de nouveaux exemples, et interagir simplement avec votre code d'une façon naturelle. Les sections qui suivent vous initient à Notebook pour que vous compreniez comment s'articule tout ce concept de répertoire.

Définir le dossier du livre

Il est avantageux pour vous d'organiser vos fichiers en vue d'y accéder plus facilement par la suite. Les fichiers de ce livre, par exemple, sont conservés dans le dossier APLN (Algorithmes pour les Nuls). Pour créer un nouveau dossier dans Notebook, procédez comme suit :

1. Choisissez Nouveau à Dossier.

Notebook crée un nouveau dossier appelé Dossier sans titre ([Figure 3-6](#)). Sachant que les fichiers apparaissent par ordre alphanumérique, il se peut que vous ne le voyiez pas tout de suite et que vous soyez obligé de faire défiler l'écran jusqu'à le voir apparaître.

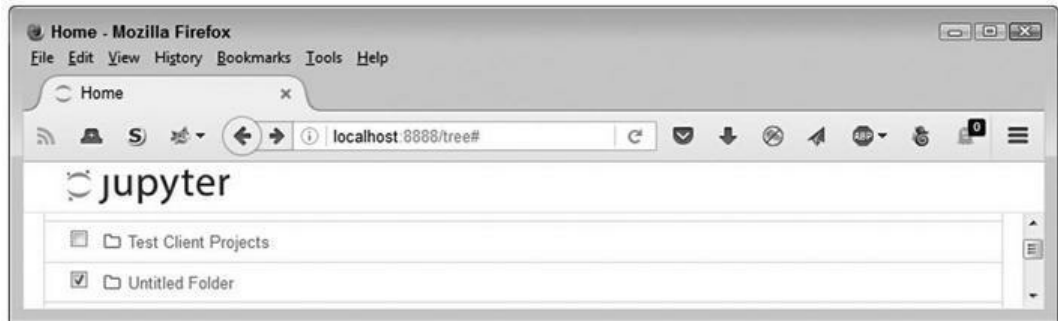


FIGURE 3-6 Les nouveaux dossiers apparaissent sous le nom de Dossier sans titre.

2. **Cochez la case à côté de l'entrée « Dossier sans titre ».**
3. **Cliquez sur Renommer en haut de la page.**

Vous voyez apparaître une boîte de dialogue « Renommer » comme celle de la [Figure 3-7](#).

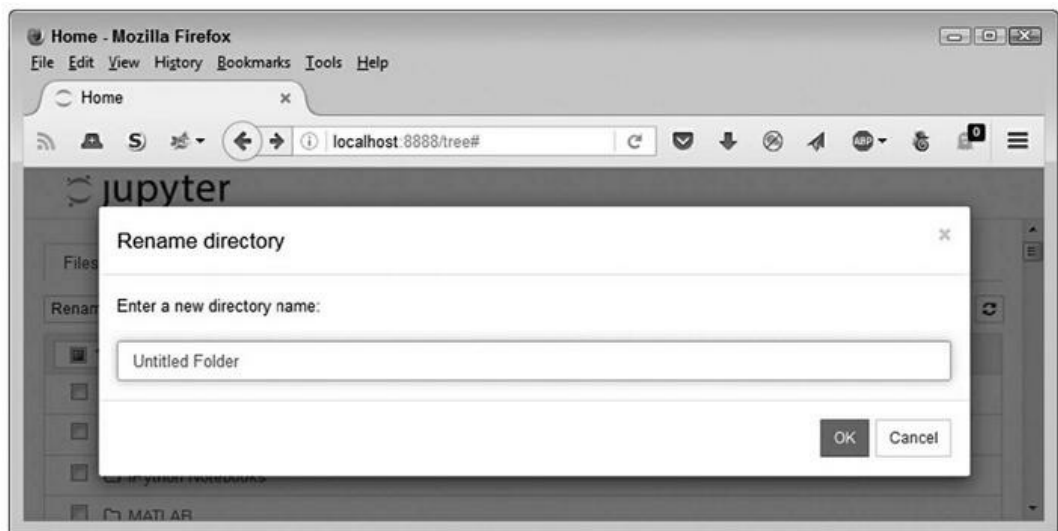


FIGURE 3-7 Renommez le dossier pour vous rappeler le type d'entrées qu'il doit contenir.

4. **Entrez A4D et cliquez sur OK.**
Notebook change le nom du dossier.
5. **Cliquez sur la nouvelle entrée A4D dans la liste.**

Notebook change l'emplacement dans lequel vous allez créer des tâches correspondant aux exercices de ce livre et l'attribue au dossier A4D.

Créer un nouveau notebook

Chaque nouveau notebook est comme un dossier. Vous pouvez placer des exemples dans ce dossier virtuel, tout comme vous classeriez des feuilles de papier dans un dossier physique. Chaque exemple apparaît dans une cellule. Dans le dossier, vous pouvez ranger également d'autres types d'éléments, mais vous découvrirez tout cela à mesure que vous avancerez dans l'utilisation de ce livre. Pour créer un nouveau notebook, procédez comme suit :

- 1. Cliquez sur Nouveau à Python (default).**

Un nouvel onglet s'ouvre dans le navigateur avec le nouveau notebook ([Figure 3-8](#)). Il convient de remarquer que le notebook contient une cellule que Notebook a mise en surbrillance pour que vous puissiez commencer à y saisir du code. Le titre du notebook est alors Sans titre. Comme un tel titre n'est pas particulièrement utile, il faut que vous le changiez.

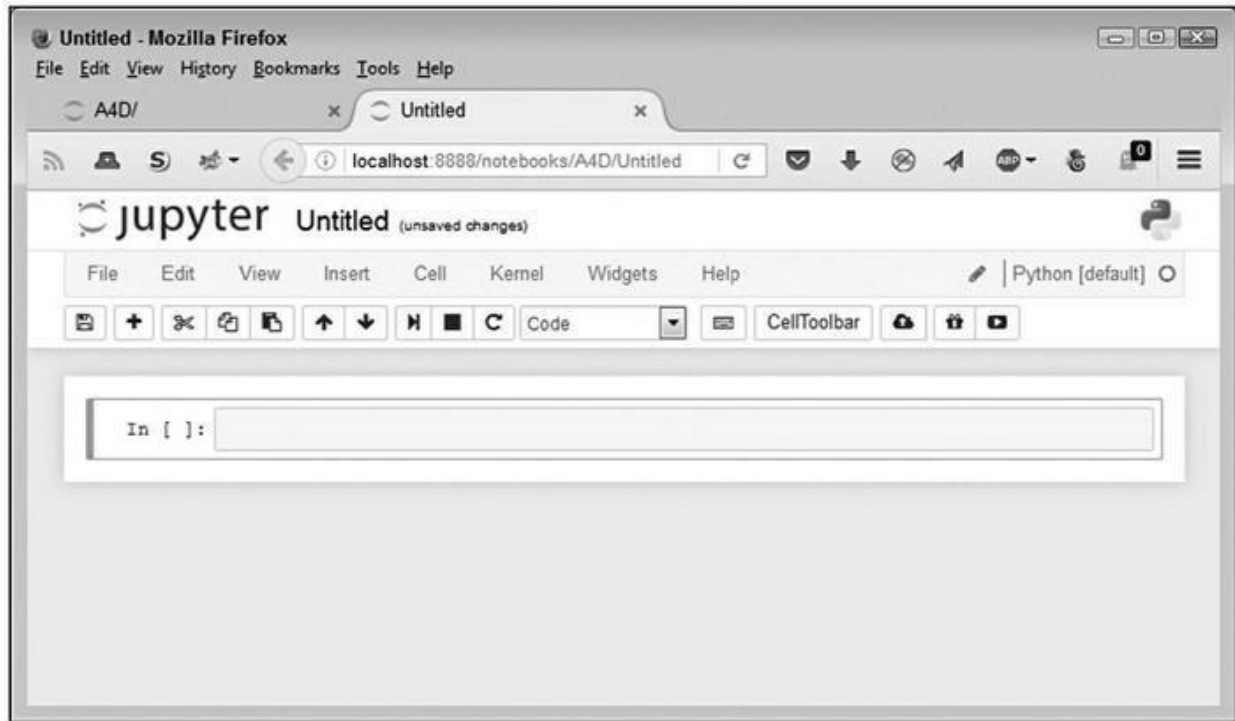


FIGURE 3-8 Un notebook contient des cellules que vous utilisez pour stocker du code.

2. Cliquez sur Sans titre sur la page.

Notebook vous demande quel nouveau nom vous voulez utiliser ([Figure 3-9](#)).

3. Tapez A4D; 03; Sample et appuyez sur Entrée.

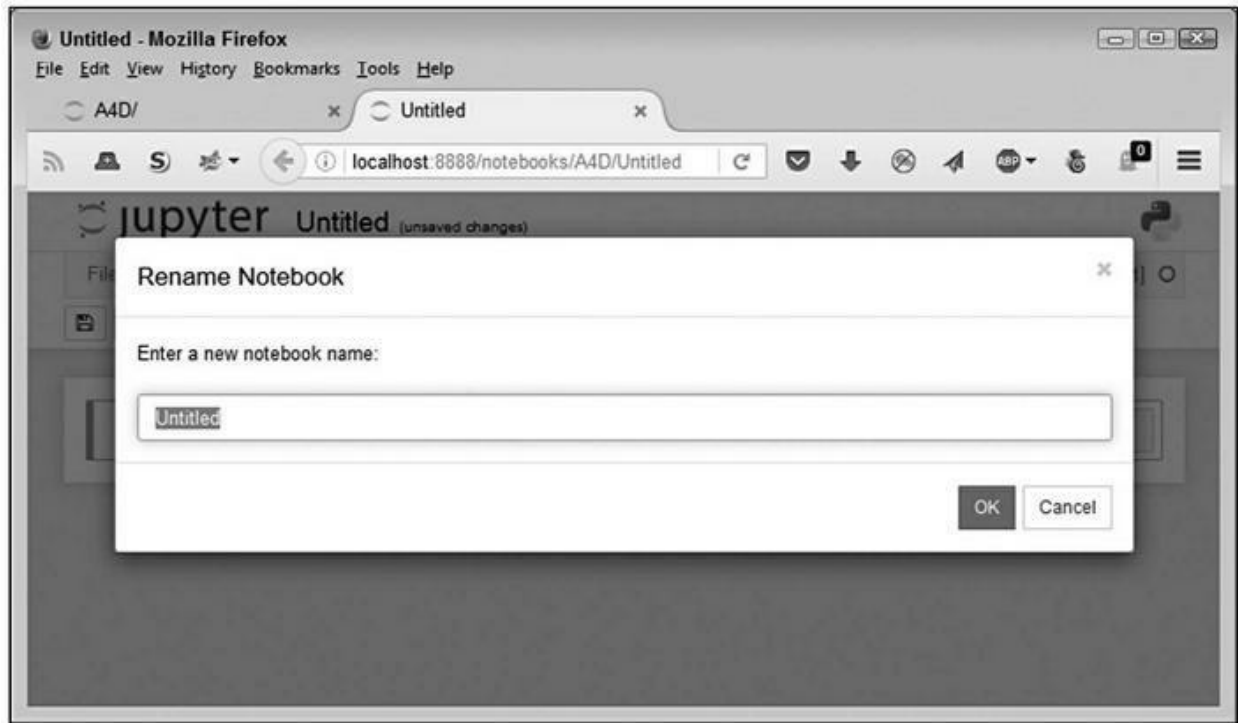


FIGURE 3-9 Donnez un nouveau nom à votre notebook.

Ce nouveau nom signifie que c'est un fichier pour *Les Algorithmes pour les Nuls*, [Chapitre 3](#), `Sample.ipynb`. Cette convention d'appellation vous permettra de différencier facilement ces fichiers des autres fichiers de votre répertoire.

Bien sûr, le notebook `Sample` est encore vide de tout contenu. Placez le pointeur dans la cellule, tapez « **Python c'est vraiment chouette !** », puis cliquez sur le bouton Exécuter (le bouton avec la flèche vers la droite sur la barre d'outils). La [Figure 3-10](#) montre le résultat, qui s'affiche dans la même cellule que le code (le code est dans une case et le résultat à l'extérieur de cette case, mais l'un et l'autre se trouvent à l'intérieur de la cellule). Toutefois, Notebook organise une séparation visuelle entre les deux, pour vous permettre de les distinguer. Notebook crée automatiquement une nouvelle cellule.

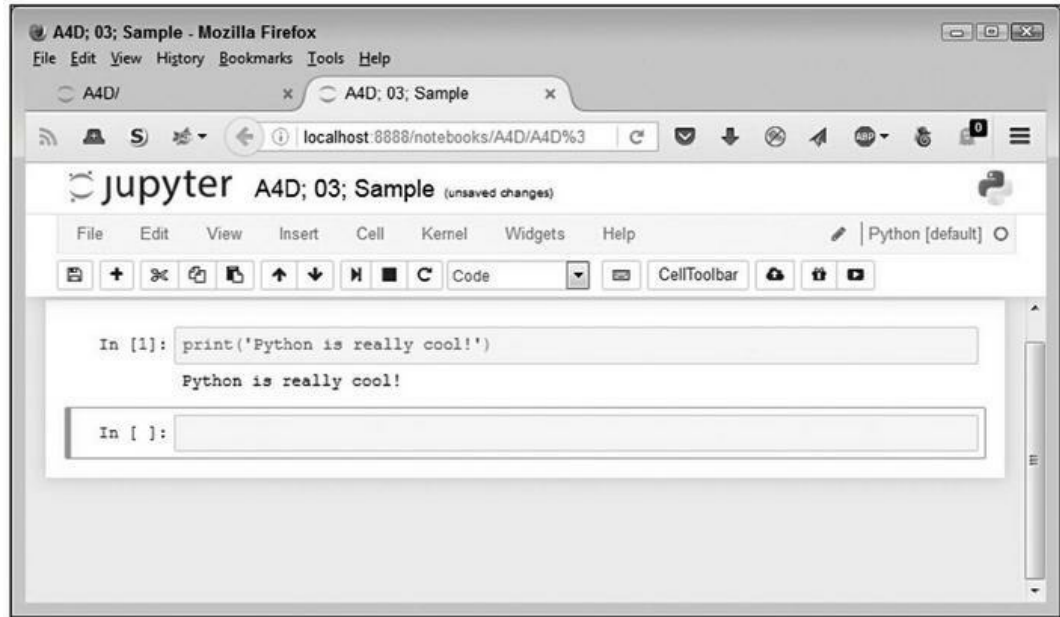


FIGURE 3-10 Notebook utilise des cellules pour stocker votre code.

Quand vous avez fini de travailler avec un notebook, il est important de le refermer. Pour cela, choisissez File → Close et Halt. Vous revenez sur la page d'accueil, sur laquelle vous pouvez constater que le notebook que vous venez de créer a été ajouté à la liste ([Figure 3-11](#)).

Exporter un notebook

Créer des notebooks et les garder pour soi n'est pas très amusant. Il arrive un moment où vous avez envie de les partager avec d'autres. Pour ce faire, vous devez exporter votre notebook du répertoire vers un fichier. Vous pouvez envoyer le fichier à quelqu'un d'autre, qui l'importera dans son répertoire.

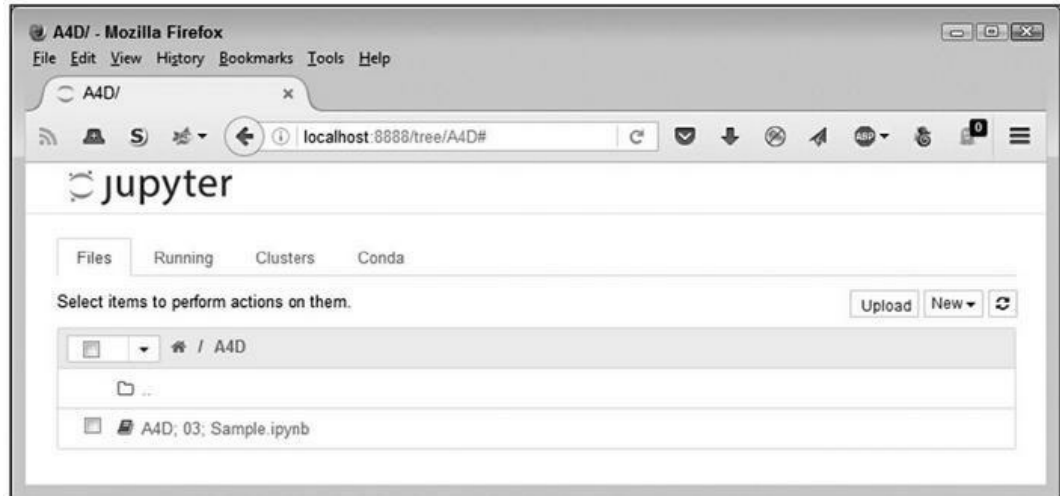


FIGURE 3-11 Tout notebook que vous créez apparaît dans la liste du répertoire.

La section précédente montre comment créer un notebook qui sera nommé A4D ; 03 ; Sample. Vous pouvez ouvrir ce notebook en cliquant sur l'entrée qui lui correspond dans la liste du répertoire. Le fichier se rouvre et vous pouvez voir à nouveau le code. Pour exporter ce code, sélectionnez File → Download As → Notebook (.ipynb). Ce qui apparaît ensuite à l'écran dépend de votre navigateur, mais en général vous voyez s'afficher une sorte de boîte de dialogue pour sauvegarder le notebook en tant que fichier. Utilisez la même méthode pour sauvegarder ce fichier IPython Notebook que pour n'importe quel autre fichier que vous sauvegarderiez en utilisant votre navigateur.

Supprimer un notebook

Il arrive que des notebooks soient périmés, ou que vous n'en ayez simplement plus besoin. Plutôt que de laisser votre répertoire se saturer progressivement de fichiers inutiles, supprimez de la liste ces notebooks dont vous ne voulez plus. Procédez comme suit :

1. **Cochez la case à côté de l'entrée A4D; 03; Sample.ipynb.**

2. Cliquez sur l'icône représentant une corbeille (Delete) en haut de la page.

Vous voyez s'afficher un message de sécurité ([Figure 3-12](#)).

3. Cliquez sur Delete.

Le fichier est supprimé de la liste.

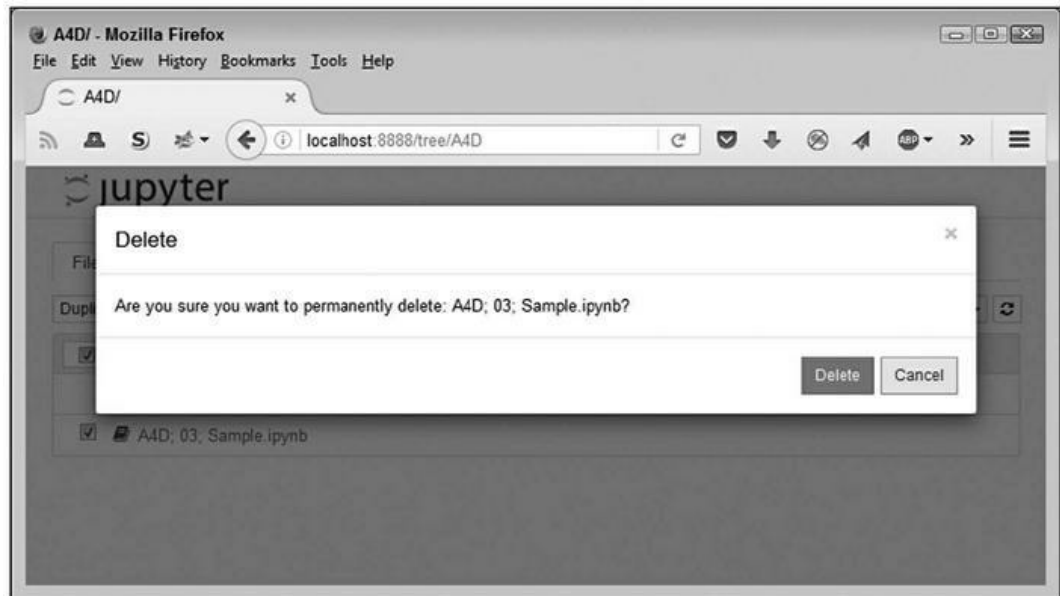


FIGURE 3-12 Notebook affiche un message d’alerte avant de supprimer un fichier du répertoire.

Importer un notebook

Pour utiliser le code source de ce livre, vous devez importer les fichiers téléchargés dans votre répertoire. Le code source se trouve dans un fichier d’archive que vous extrayez et que vous copiez quelque part sur votre disque dur. L’archive comporte une liste de fichiers en .ipynb (IPython Notebook) qui contiennent le code source pour ce livre (pour plus de détails sur le téléchargement du code source, consulter l’introduction). Pour importer ces fichiers dans votre répertoire, procédez comme suit :

1. Cliquez sur Upload en haut de la page.

Le contenu de l'écran dépend de votre navigateur. Dans la plupart de cas, vous voyez s'afficher une sorte de boîte de dialogue pour télécharger un fichier, donnant accès aux fichiers qui se trouvent sur votre disque dur.

2. **Accédez au répertoire contenant les fichiers que vous voulez importer dans Notebook.**
3. **Sélectionnez le ou les fichiers à importer, et cliquez sur le bouton Open (Ouvrir), ou autre bouton similaire, pour lancer le téléchargement.**

Le fichier vient s'ajouter à la liste de téléchargement ([Figure 3-13](#)). Il ne fait pas encore partie du répertoire : vous l'avez simplement sélectionné pour qu'il soit téléchargé.

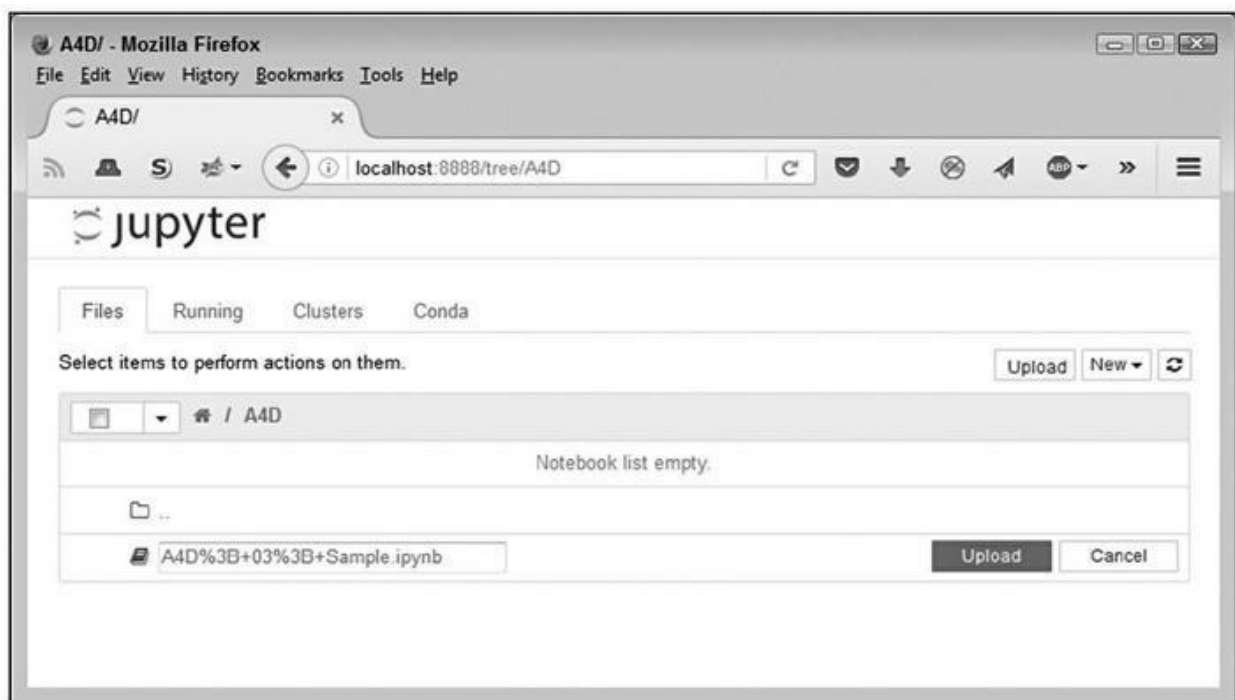


FIGURE 3-13 Les fichiers que vous voulez ajouter au répertoire apparaissent en tant que partie d'une liste de téléchargements constituée d'un ou plusieurs noms de fichiers.



Quand vous exportez un fichier, Notebook convertit les éventuels caractères spéciaux et les met sous une forme que votre système pourra gérer plus facilement. La [Figure 3-](#)

[13](#) illustre cette conversion. Le point-virgule apparaît sous la forme %3B, et l'espace sous la forme + (signe plus). Dans Notebook, vous devez remplacer ces caractères pour voir le titre s'afficher normalement.

4. Cliquez sur Upload (Télécharger).

Notebook place le fichier dans le répertoire, et vous pouvez dès lors vous en servir.

Comprendre les jeux de données utilisés dans ce livre

Ce livre utilise un certain nombre de jeux de données, qui sont tous présents dans le module Scikit-learn. Ces jeux de données mettent en pratique plusieurs manières d'interagir avec les données, et vous allez les utiliser dans les exemples pour effectuer des tâches variées. La liste suivante donne un bref aperçu de la fonction utilisée pour importer chacun des jeux de données dans votre code Python :

- » `load_boston()` : Analyse de régression avec le jeu de données sur les prix des logements à Boston.
- » `load_iris()` : Classification avec le jeu de données Iris.
- » `load_diabetes()` : Régression avec le jeu de données sur le diabète.
- » `load_digits([n_class])` : Classification avec le jeu de données Digits.
- » `fetch_20newsgroups(subset='train')` : Données de 20 newsgroups.
- » `fetch_olivetti_faces()` : Jeux de données Olivetti sur les visages, d'AT&T.

La technique de chargement des jeux de données est la même pour tous les exemples. L'exemple suivant montre comment charger le

jeu de données sur les prix des logements à Boston. Vous trouverez le code dans le notebook A4D ; 03 ; Dataset Load.ipynb.

```
from sklearn.datasets import load_boston
Boston = load_boston()
print(Boston.data.shape)
```

(506, 13)

Pour voir comment fonctionne ce code, cliquez sur le bouton Exécution. Le résultat de l'appel de print() est (506, 13). La [Figure 3-14](#) montre le résultat.

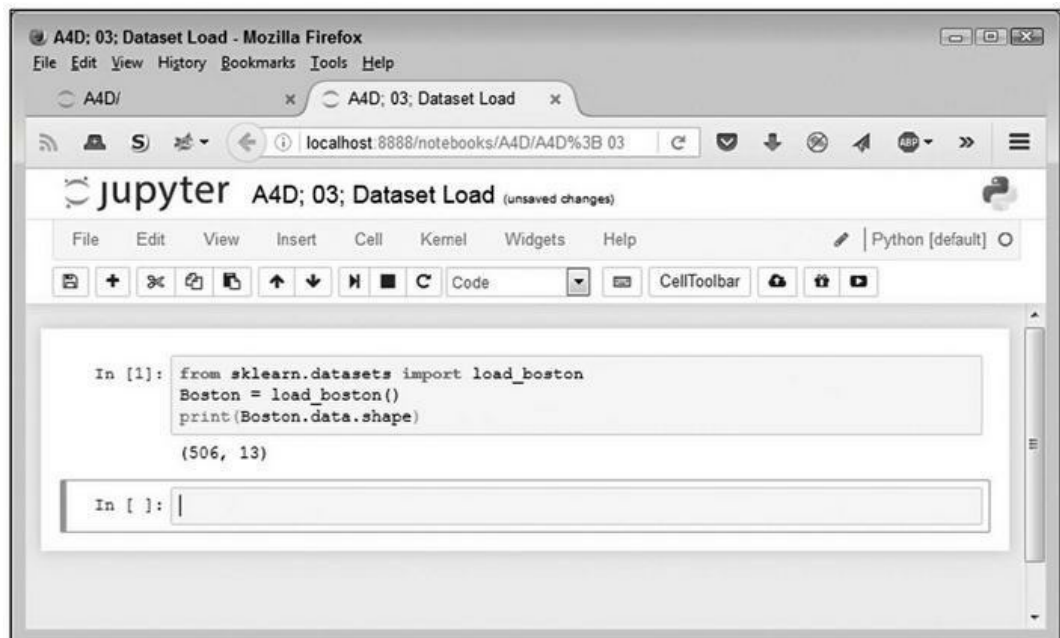


FIGURE 3-14 L'objet Boston contient le jeu de données chargé.

Chapitre 4

Utiliser Python pour la programmation algorithmique

DANS CE CHAPITRE

- » Exécuter des tâches numériques et logiques
 - » Travailler avec des chaînes
 - » Exécuter des tâches avec des dates
 - » Compléter le code par des fonctions
 - » Préparer des décisions et répéter des étapes
 - » Gérer les données en mémoire
 - » Lire les données dans des objets de stockage
 - » Trouver plus vite les données en utilisant des dictionnaires
-

Une recette de cuisine est une sorte d'algorithme. En effet, elle vous sert à préparer quelque chose de bon par étapes successives (jusqu'à ce que vous puissiez apaiser votre fringale). On peut trouver différentes manières de créer une série d'étapes en vue de résoudre un problème. Il y a abondance de procédures et de descriptions variées, mais il s'agit toujours de décrire une série d'étapes destinée à résoudre un problème. Cette série d'étapes n'est pas nécessairement concrète. La notation mathématique permet de présenter une série d'étapes pour résoudre un problème numérique, mais pour beaucoup de gens,

tous ces symboles constituent un langage mystérieux et rarement compréhensible. Un langage informatique permet de traduire ce langage peu accessible sous une forme plus concrète, en l'occurrence, sous forme d'énoncés proches de l'anglais. On peut ainsi proposer une méthode de résolution de problèmes plus accessible à la plupart des gens.

Le chapitre précédent, c'est-à-dire le [Chapitre 3](#), vous explique comment installer une copie de Python afin de pouvoir travailler sur les exemples contenus dans ce livre. Du début à la fin de ce livre, vous allez utiliser Python pour résoudre des problèmes numériques à l'aide d'algorithmes pouvant être aussi représentés dans une notation mathématique. Si ce livre utilise un langage de programmation, c'est pour remplacer tous ces symboles abstraits aux formes parfois étranges par un langage plus compréhensible par un large public et permettre à ce public de résoudre des problèmes du monde réel.

Avant de pouvoir utiliser Python pour exécuter des tâches à l'aide d'algorithmes, vous avez besoin d'acquérir au moins des notions de son fonctionnement. Ce chapitre n'est pas destiné à faire de vous un spécialiste de Python, mais à vous présenter assez d'informations pour que vous puissiez comprendre le code exemple, avec l'aide du commentaire fourni. Les différentes sections vous expliquent, de façon concrète, comment Python exécute des tâches. Il faut que vous sachiez, par exemple, comment Python traite divers types de données, afin de pouvoir déterminer ce que le code exemple fait avec ces données. Dans les trois premières sections de ce chapitre, vous trouverez les bases nécessaires pour travailler avec des données numériques, des données logiques, des chaînes et des dates.

Imaginons un livre de recettes de cuisine, ou même un livre quelconque, proposant des tâches à accomplir par étapes successives mais expliquant tout cela sous la forme d'un long texte ininterrompu. Il deviendrait impossible d'y trouver une recette (ou une procédure) en particulier, et un tel livre n'aurait aucune utilité. En réalité, personne n'irait rédiger un tel livre. La quatrième section de ce chapitre est consacrée aux fonctions,

qu'on peut assimiler aux recettes d'un livre de cuisine. Vous pouvez combiner des fonctions pour créer un programme, tout comme vous pourriez combiner des recettes pour préparer tout un repas.

Les quatre sections suivantes présentent plusieurs manières de gérer les données, c'est-à-dire de les lire, de les enregistrer, de les modifier et de les supprimer à volonté. Vous avez aussi besoin de savoir comment formuler des décisions et que faire quand vous devez exécuter plus d'une fois la même série d'étapes. Les données sont une ressource, tout comme la farine, le sucre et les autres ingrédients sont les ressources que vous utilisez quand vous appliquez une recette de gâteau. Pour intégrer les différents types de données dans une application résolvant le problème proposé par un algorithme, des techniques différentes sont nécessaires. Ces sections vous expliquent les différentes manières de manipuler les données et de les utiliser pour résoudre des problèmes.

Travailler avec des nombres et des règles logiques

Interagir avec des algorithmes, c'est manier différentes sortes de données, mais surtout des nombres. On utilise aussi des valeurs logiques pour programmer des décisions concernant les données utilisées. Ainsi, par exemple, il se peut que vous ayez besoin de savoir si deux valeurs sont égales, ou si une valeur est supérieure à une autre. Python prend en charge les types de valeurs numériques et logiques suivants :

- » Tout nombre sans décimale est un *entier*. La valeur 1, par exemple, est un entier, mais la valeur 1,0 qui comporte une décimale (bien qu'elle soit nulle) n'est pas considérée ici comme un entier. Les entiers sont représentés par le type de donnée `int`. Sur la plupart des plateformes, on peut stocker des nombres compris entre $-9\,223\,372\,036\,854\,775\,808$ et $9\,223\,372\,036\,854\,775\,807$ dans les variables de type `int` (il

s'agit de la valeur maximum que peut prendre une variable codée sur 64 bits).

- » Tout nombre comportant une partie décimale est une *valeur à virgule flottante*. Par exemple, 1,0 comporte une décimale. Les gens confondent souvent les entiers et les nombres à virgule flottante, mais la différence est facile à retenir. Quand vous voyez une décimale, il s'agit d'une valeur à virgule flottante. Python stocke ces valeurs en utilisant le type de donnée float. Sur la plupart des plateformes, la valeur maximum que peut prendre une variable à virgule flottante est $\pm 1,7976931348623157 \times 10^{308}$ et la valeur minimum est $\pm 2,2250738585072014 \times 10^{-308}$.
- » Un *nombre complexe* est constitué d'un nombre réel et d'un nombre imaginaire, associés l'un à l'autre. Au cas où vous auriez tout oublié de vos cours sur les nombres complexes, vous pouvez consulter la page <http://www.mathsisfun.com/numbers/complex-numbers.html>. La partie imaginaire d'un nombre complexe apparaît toujours suivie d'un j (dans le contexte anglo-saxon de l'informatique, bien qu'en mathématiques nous ayons plutôt l'habitude de la faire précéder d'un i – N.d.T.). Par conséquent, si vous voulez utiliser un nombre complexe dont la partie réelle est 3 et la partie imaginaire 4, vous procéderez à l'affectation suivante : `myComplex = 3 + 4j`.
- » Les arguments logiques fonctionnent grâce à des valeurs booléennes, du nom de George Boole. Dans Python, pour les valeurs booléennes, on utilise le type bool. Une variable de ce type ne peut prendre que deux valeurs, True et False (c'est-à-dire Vrai et Faux). Pour affecter une valeur à une variable, vous pouvez utiliser les mots-clés True et False, ou bien créer une expression définissant une idée logique équivalente à True ou à False. Ainsi, par exemple, si vous écrivez `myBool = 1 > 2`, cette expression est équivalente à False car 1 n'est bien évidemment pas supérieur à 2.

Maintenant que vous avez acquis ces bases, il est temps de voir comment manier ces types de données. Les paragraphes qui suivent donnent un bref aperçu de la manière dont vous pouvez travailler avec les données numériques et logiques dans Python.

Procéder à des affectations de variables

Quand vous utilisez des applications, vous stockez l'information dans des variables. Une *variable* est comme une boîte de rangement. Chaque fois que vous voulez utiliser l'information, c'est en utilisant une variable que vous y accédez. Quand vous voulez stocker une nouvelle information, vous la faites entrer dans une variable. Modifier une information, c'est accéder tout d'abord à la variable concernée, puis stocker la nouvelle valeur dans cette variable. Tout comme on range des objets dans des boîtes dans le monde réel, on stocke ici des éléments dans des variables. On *affecte* des données à des variables en utilisant des *opérateurs d'affectation* (des symboles particuliers qui indiquent de quelle manière les données doivent être stockées). Le Tableau 4-1 présente les opérateurs d'affectation supportés par Python.

TABLEAU 4-1 Opérateurs d'affectation de Python

Opérateur	Description	Exemple
=	Affecte la valeur de l'opérande droit dans l'opérande gauche.	MyVar = 5 a pour effet de rendre le contenu de MyVar égal à 5
+=	Ajoute la valeur de l'opérande droit à l'opérande gauche et place le résultat dans l'opérande gauche.	MyVar += 2 a pour effet de rendre le

		contenu de MyVar égal à 7
-=	Soustrait la valeur de l'opérande droit à l'opérande gauche et place le résultat dans l'opérande gauche.	MyVar -= 2 a pour effet de rendre le contenu de MyVar égal à 3
*=	Multiplie la valeur de l'opérande droit par la valeur de l'opérande gauche et place le résultat dans l'opérande gauche.	MyVar *= 2 a pour effet de rendre le contenu de MyVar égal à 10
/=	Divise la valeur de l'opérande gauche par la valeur de l'opérande droit et place le résultat dans l'opérande gauche.	MyVar /= 2 a pour effet de rendre le contenu de MyVar égal à 2,5
%=	Divise la valeur de l'opérande gauche par la valeur de l'opérande droit et place le reste dans l'opérande gauche.	MyVar %= 2 a pour effet de rendre le contenu de MyVar égal à 1
**=	Détermine la valeur exponentielle de l'opérande gauche quand elle est élevée à une puissance égale à la valeur de l'opérande droit et place le reste dans l'opérande gauche.	MyVar **= 2 a pour effet de rendre le contenu de MyVar égal à 25
//=	Divise la valeur de l'opérande gauche par la valeur de l'opérande droit et place le résultat entier dans l'opérande gauche.	MyVar //= 2 a pour effet de rendre le

Un peu d'arithmétique

Stocker l'information dans des variables permet de la rendre facilement accessible. Cependant, de façon pratique, l'utilisation d'une variable consiste généralement à effectuer sur cette variable une opération arithmétique. Python supporte les opérateurs arithmétiques couramment utilisés dans les calculs effectués à la main (voir Tableau 4-2).

TABLEAU 4-2 Opérateurs arithmétiques de Python

Opérateur	Description	Exemple
+	Additionne deux valeurs.	5+2=7
-	Soustrait l'opérande droit de l'opérande gauche.	5 - 2=3
*	Multiplie l'opérande droit par l'opérande gauche.	5 * 2 = 10
/	Divise l'opérande gauche par l'opérande droit.	5 / 2 = 2.5
%	Divise l'opérande gauche par l'opérande droit et donne le reste.	5 % 2=1
**	Calcule la valeur exponentielle de l'opérande droit par l'opérande gauche.	5 ** 2 = 25
//	Effectue la division, entière, consistant à diviser l'opérande gauche par l'opérande droit pour ne produire que la partie entière.	5 // 2 = 2

Il arrive que l'on ait besoin d'effectuer une opération avec une seule variable. Python supporte un certain nombre d'*opérateurs*

unaires, c'est-à-dire applicables à une variable unique. Ils sont présentés dans le Tableau 4-3.

TABLEAU 4-3 Opérateurs unaires de Python

Opérateur	Description	Exemple
~	Inverse les bits dans un nombre, de telle sorte que tous les bits 0 deviennent des bits 1, et inversement.	~4 retourne le résultat -5
-	Inverse le signe, de telle sorte qu'une valeur positive devienne négative et inversement.	-(4) retourne le résultat -4 et -(-4) retourne le résultat 4
+	Est prévu simplement par souci d'exhaustivité : retourne la valeur de l'input.	+4 retourne le résultat 4

Les ordinateurs peuvent exécuter aussi d'autres types de tâches mathématiques, compte tenu de la façon dont fonctionne un processeur. Il est important de ne pas oublier qu'un ordinateur stocke les données sous forme de séries de bits. Python vous permet d'accéder à chacun de ces bits grâce aux opérateurs de bits (Tableau 4-4).

TABLEAU 4-4 Opérateurs sur les bits de Python

Opérateur	Description	Exemple
& (et)	Détermine si les deux bits dans les deux opérandes ont la valeur Vrai et si c'est le cas, affecte la valeur Vrai au bit résultant.	0b1100 & 0b0110 = 0b0100
(ou)	Détermine si un des deux bits dans les deux opérandes a la valeur Vrai et si c'est le cas, affecte la valeur Vrai au bit résultant.	0b1100 0b0110 = 0b1110

$\wedge$ (ou exclusif)	Détermine si un seul des deux bits dans les deux opérandes a la valeur Vrai et si c'est le cas, affecte la valeur Vrai au bit résultant. Si les deux bits ont chacun la valeur Vrai ou chacun la valeur Faux, la valeur résultante est Faux.	0b1100 $\wedge$ 0b0110 = 0b1010
$\sim$ (complément à 1)	Calcule le complément à un d'un nombre.	$\sim$ 0b1100 = -0b1101 $\sim$ 0b0110 = -0b0111
$\ll$ (décalage à gauche)	Décale vers la gauche les bits de l'opérande de gauche d'un nombre de positions égal à la valeur de l'opérande de droite. Tous les nouveaux bits sont mis à 0 et tous les bits qui se retrouvent au-delà de l'extrémité de l'opérande sont perdus.	0b00110011 $\ll$ 2 = 0b11001100
$\gg$ (décalage à droite)	Décale vers la droite les bits de l'opérande de droite d'un nombre de positions égal à la valeur de l'opérande de gauche. Tous les nouveaux bits sont mis à 0 et tous les bits qui se retrouvent au-delà de l'extrémité de l'opérande sont perdus.	0b00110011 $\gg$ 2 = 0b00001100

Comparer les données à l'aide d'expressions booléennes

L'utilisation de l'arithmétique pour modifier le contenu des variables est une forme de manipulation des données. Pour déterminer l'effet d'une manipulation de données, l'ordinateur doit comparer l'état courant de la variable à son état initial ou à l'état d'une valeur connue. Dans certains cas, il est aussi nécessaire de distinguer l'état d'un input de l'état d'un autre. Toutes ces opérations consistent à examiner la relation entre deux

variables, par conséquent les opérateurs résultants sont des opérateurs relationnels (Tableau 4-5).

TABEAU 4-5 Opérateurs relationnels de Python

Opérateur	Description	Exemple
==	Détermine si deux valeurs sont égales. Il convient de remarquer que cet opérateur relationnel utilise deux signes égal. Un certain nombre de développeurs font souvent l'erreur de n'utiliser qu'un signe égal, avec pour résultat que la valeur d'un opérande est affectée à un autre opérande.	1 == 2 est Faux
!=	Détermine si deux valeurs sont différentes. Certaines versions plus anciennes de Python vous permettaient d'utiliser l'opérateur <> à la place de l'opérateur !=, mais dans les versions actuelles, utiliser l'opérateur <> entraîne une erreur.	1 != 2 est Vrai
>	Vérifie que la valeur de l'opérande de gauche est supérieure à la valeur de l'opérande de droite.	1 > 2 est Faux
<	Vérifie que la valeur de l'opérande de gauche est inférieure à la valeur de l'opérande de droite.	1 < 2 est Vrai
>=	Vérifie que la valeur de l'opérande de gauche est supérieure ou égale à la valeur de l'opérande de droite.	1 >= 2 est Faux
<=	Vérifie que la valeur de l'opérande de gauche est inférieure ou égale à la valeur de l'opérande de droite.	1 <= 2 est Vrai

Un opérateur relationnel ne peut pas toujours effectuer une comparaison suffisamment précise entre deux valeurs. Ainsi, par exemple, il est parfois nécessaire de tester une condition

impliquant deux comparaisons distinctes, comme `MonAge > 40` et `MaTaille < 74`. Lorsqu'il faut ajouter des conditions à la comparaison, un opérateur logique comme ceux du [Tableau 4-6](#) est nécessaire.

TABLEAU 4-6 Opérateurs logiques de Python

Opérateur	Description	Exemple
<code>and</code>	Détermine si les deux opérandes sont True.	<code>True and True</code> est True <code>True and False</code> est False <code>False and True</code> est False <code>False and False</code> est False
<code>or</code>	Détermine si un des deux opérandes est True.	<code>True or True</code> est True <code>True or False</code> est True <code>False or True</code> est True <code>False or False</code> est False
<code>not</code>	Inverse la valeur d'un unique opérande. Une valeur True devient False et une valeur False devient True.	<code>not True</code> est False

not False
est True

L'ordinateur ordonne les comparaisons en donnant à certains opérateurs la priorité sur d'autres. L'ordre des opérateurs est ce que l'on appelle aussi en mathématique la *priorité des opérations*. Le [Tableau 4-7](#) présente l'ordre des opérateurs courants de Python, y compris quelques-uns qui n'ont pas été abordés dans tout ce qui précède. Quand vous effectuez des comparaisons, tenez toujours compte de l'ordre des opérateurs, faute de quoi vos suppositions concernant une comparaison auraient de bonnes chances d'être erronées.

TABLEAU 4-7 Ordre des opérateurs de Python

Opérateur	Description
()	On utilise des parenthèses pour regrouper des expressions et éviter que s'applique l'ordre de priorité par défaut. En d'autres termes, on fait en sorte qu'une opération de plus faible priorité (comme l'addition) devienne prioritaire par rapport à une opération de plus forte priorité (comme la multiplication).
**	L'exponentiation élève la valeur de l'opérande de gauche à une puissance égale à la valeur de l'opérande de droite.
~+-	Les opérateurs unaires interagissent avec une unique variable ou expression.
* / % //	Multiplication, division, modulo, et division entière.
+-	Addition et soustraction.
>> <<	Décalage de bits à droite et à gauche.
&	Le ET (AND) bit à bit.
^	Le OU (OR) exclusif bit à bit et le OU (OR) non exclusif bit à bit.

<= < > >=	Les opérateurs de comparaison.
== !=	Les opérateurs d'égalité.
= %= /= //= -= += *= **=	Les opérateurs d'affectation.
is	
is not	Les opérateurs d'identité.
in	
not in	Les opérateurs d'appartenance.
not or and	Les opérateurs logiques.

Créer et utiliser des chaînes

Parmi tous les types de données, les chaînes sont ce que les humains comprennent le plus facilement, alors que les ordinateurs ne les comprennent pas du tout. Une *chaîne* est simplement un regroupement quelconque de caractères placé entre guillemets. Ainsi, par exemple, `myString = "Python est un langage épatant"` affecte une chaîne de caractères à la variable `myString`.



La principale raison d'utiliser des chaînes quand on travaille sur les algorithmes est de faciliter les interactions des utilisateurs, soit sous forme de requêtes en guise d'input, soit comme moyen de rendre l'output plus facile à comprendre. Vous pouvez aussi procéder à une analyse des chaînes de données, mais l'ordinateur n'a pas besoin que des chaînes interviennent dans la succession des étapes de l'algorithme pour aboutir à une solution au problème étudié. En réalité, l'ordinateur ne connaît pas les lettres. Chaque lettre que vous utilisez est représentée par un nombre en mémoire. La lettre A, par exemple, est en réalité le nombre 65. Pour le constater de vos propres yeux, saisissez `ord("A")` dans l'invite de Python et appuyez sur Entrée. Vous verrez s'afficher comme résultat 65. Vous pouvez convertir n'importe quelle lettre

en son équivalent numérique en utilisant la commande `ord()`.

Sachant que l'ordinateur ne comprend pas réellement les chaînes, mais que les chaînes sont très utiles quand on développe des applications, vous aurez parfois besoin de convertir une chaîne en nombre. Pour ce faire, vous pouvez utiliser les commandes `int()` et `float()`. Ainsi, par exemple, en saisissant **`myInt = int("123")`** et en appuyant sur Entrée à l'invite de Python, vous créez un `int` appelé `myInt` et contenant la valeur 123.



Vous pouvez tout aussi bien convertir des nombres en chaînes à l'aide de la commande `str()`. Ainsi, en saisissant **`myStr = str(1234.56)`** et en appuyant sur Entrée, vous créez une chaîne contenant la valeur "1234.56" et vous l'affectez à `myStr`. En un mot, vous pouvez changer des nombres en chaînes et des chaînes en nombres en toute facilité. Les chapitres qui suivent montrent comment ces conversions rendent faisables des tâches qui pourraient paraître impossibles.

Comme pour les nombres, vous pouvez utiliser des opérateurs spéciaux avec les chaînes (et avec un certain nombre d'objets). Les *opérateurs d'appartenance* vous permettent de déterminer si une chaîne contient un contenu spécifique. Le [Tableau 4-8](#) montre ces opérateurs.

TABLEAU 4-8 Opérateurs d'appartenance de Python

Opérateur	Description	Exemple
<code>in</code>	Détermine si la valeur de l'opérande de gauche apparaît dans la séquence contenue dans l'opérande de droite.	"Hello" dans "Hello Goodbye" est True
<code>not in</code>	Détermine si la valeur de l'opérande de gauche est absente dans la séquence contenue dans l'opérande de droite.	"Hello" not in "Hello Goodbye" est False

Par ailleurs, de ce qui est exposé dans cette section, il ressort clairement que vous avez besoin de connaître le type de données que contiennent ces variables. Pour ce faire, vous devez utiliser les opérateurs d'identité (Tableau 4-9).

TABEAU 4-9 Opérateurs d'identité de Python

Opérateur	Description	Exemple
is	Est évalué à True quand le type de valeur ou l'expression dans l'opérande de droite correspond au type de l'opérande de gauche.	type(2) is int est True
is not	Est évalué à True quand le type de valeur ou l'expression dans l'opérande de droite ne correspond pas au type de l'opérande de gauche.	type(2) is not int est False

LANCER IPYTHON

La plus grande partie de ce livre fait référence à Jupyter Notebook (voir [Chapitre 3](#)) parce que cet outil fournit des méthodes pour créer, gérer, et interagir avec des exemples complexes de codage. Cependant, vous avez parfois besoin d'un environnement interactif simple pour effectuer des tests rapides, et c'est là l'orientation donnée à ce chapitre. Anaconda s'accompagne de deux environnements répondant à ce besoin : IPython et Jupyter QT Console. Des deux, IPython est le plus simple d'utilisation, mais les deux environnements offrent des fonctionnalités similaires. Pour lancer IPython, cliquez simplement sur son entrée dans le dossier Anaconda3, dans votre système. Sous Windows, par exemple, sélectionnez Démarrer → Tous les programmes → Anaconda3 → IPython. Vous pouvez aussi lancer IPython à partir d'une fenêtre de console ou de terminal en saisissant IPython et en appuyant sur Entrée.

Interagir avec des dates

Les dates et les durées sont des éléments avec lesquels la plupart des gens ont l'habitude de travailler. Dans notre société, pratiquement tout est basé sur le moment à consacrer à une tâche et sur la durée nécessaire à son exécution. Nous fixons des rendez-vous et nous planifions des événements à des dates précises, et en prévoyant des durées précises. Notre journée se déroule pour l'essentiel de façon minutée. Quand on travaille avec des algorithmes, le moment et la durée d'une étape donnée dans une séquence peuvent avoir autant d'importance que les déterminants de cette étape et le résultat de son exécution. Les algorithmes utilisent les dates et les durées pour organiser les données, ce qui nous permet de mieux comprendre les données et le résultat obtenu.

Compte tenu de cette importance que nous donnons naturellement au temps, il est judicieux d'étudier la manière dont Python gère les interactions avec les dates et les durées (et surtout, la manière dont il stocke ces valeurs en vue d'une utilisation ultérieure). Comme pour tout le reste, l'ordinateur ne comprend que les nombres : pour lui, les dates et les durées n'existent pas réellement. C'est l'algorithme, et non l'ordinateur, qui se sert des dates et des durées pour organiser la série d'étapes qui résoudra le problème.



Pour pouvoir utiliser des dates et des durées, vous devez recourir à une commande spéciale, `import datetime`. Techniquement, c'est ce que l'on appelle *importer un module*. Pour l'instant, ne vous souciez pas de savoir comment fonctionne cette commande : utilisez-la simplement à volonté, pour gérer les dates et les durées. Les ordinateurs comportent toujours une horloge, mais les horloges sont en fait destinées aux utilisateurs humains. Il est vrai que les logiciels sont parfois dépendants de l'horloge, mais là encore, il s'agit surtout des besoins des utilisateurs, et non de ce qui pourrait être nécessaire pour l'ordinateur. Pour obtenir l'heure, il vous suffit de saisir **`datetime.datetime.now()`** et d'appuyer sur Entrée. La date complète et l'heure, données par l'horloge de l'ordinateur, s'affichent comme suit : `datetime.datetime(2016, 12, 20, 10, 37, 24, 460099)`.

Vous aurez peut-être remarqué que la date et l'heure sont un peu difficiles à lire sous le format existant. Supposons que vous vouliez obtenir simplement la date du jour, sous un format lisible. Pour ce faire, vous allez accéder simplement à la partie date de l'output et la convertir en chaîne. Saisissez `str(datetime.datetime.now().date())` et appuyez sur Entrée. Vous obtenez un résultat un peu plus exploitable, par exemple '2016-12-20'.

Il est intéressant de savoir que Python offre aussi une commande `time()` pour obtenir l'heure. Vous pouvez obtenir des valeurs séparées pour les différentes composantes de la date et de l'heure en utilisant les valeurs `day`, `month`, `year`, `hour`, `minute`, `second` et `microsecond`. Les chapitres qui suivent vous expliquent comment utiliser ces différents éléments temporels pour faire plus facilement fonctionner les algorithmes.

Créer et utiliser des fonctions

Chacune des étapes d'un algorithme nécessite normalement une ligne unique de code Python : une instruction, quasiment en anglais, indiquant à l'ordinateur comment faire progresser d'une étape la solution du problème. En combinant un certain nombre de lignes de code, on obtient le résultat désiré. Il est parfois nécessaire de répéter les instructions avec des données différentes, et dans certains cas le code devient si long qu'il est difficile de garder en mémoire ce que fait chacune des parties de l'ensemble. Les fonctions sont des outils d'organisation grâce auxquels le code reste propre et ordonné. Par ailleurs, les fonctions facilitent la réutilisation des instructions déjà créées, au besoin avec d'autres données. Cette section vous explique tout sur les fonctions. Plus important, dans cette section, vous commencez à créer vos premières vraies applications, tout comme en créent les développeurs professionnels.

Créer des fonctions réutilisables

Vous ouvrez votre penderie, vous en sortez un pantalon et une chemise, vous en retirez les étiquettes et vous vous habillez. Le soir, après vous être déshabillé, vous jetez les vêtements à la poubelle. Hum ! Voilà qui est très inhabituel. La plupart des gens lavent leurs vêtements, les font sécher, puis les rangent dans l'attente de les porter à nouveau. Les fonctions aussi sont réutilisables. Qui a envie de répéter un certain nombre de fois les mêmes tâches ? Ce serait monotone et lassant. Quand vous créez une fonction, vous définissez une série de lignes de code que vous pourrez utiliser encore et encore pour exécuter une même tâche. Pour que l'ordinateur exécute la tâche en question, il vous suffira de lui dire quelle fonction il doit utiliser. Il exécutera sans discuter chaque instruction que comporte la fonction, et il le fera chaque fois que vous le lui demanderez.



Quand vous écrivez un programme dans le déroulement duquel doit intervenir une fonction déjà existante, ce programme est appelé le *programme appelant*, car il appelle la fonction pour que celle-ci exécute une tâche donnée. Les informations utilisées par la fonction proviennent en grande partie du programme appelant. Celui-ci doit transmettre des informations à la fonction, et la fonction retourne des informations au programme appelant.

Il fut un temps où les programmes informatiques ignoraient le concept de réutilisabilité. Les développeurs devaient à chaque fois réinventer le même code. Cependant, il n'a pas fallu longtemps pour que quelqu'un ait l'idée d'introduire les fonctions, et ce concept a évolué au cours du temps pour devenir de plus en plus flexible. Vous pouvez créer une fonction pour exécuter ce que vous voulez. Les applications intègrent nécessairement la réutilisabilité du code, qui offre les avantages suivants :

- » Réduction du temps de développement
- » Réduction des risques d'erreur de programmation
- » Fiabilité accrue de l'application

- » Possibilité pour des groupes entiers d'utilisateurs de bénéficier du travail d'un programmeur
- » Code plus facile à comprendre
- » Efficacité accrue de l'application

Du fait de leur réutilisabilité, les fonctions effectuent toute une liste de choses pour les applications. À travers les exemples proposés dans ce livre, vous vous rendrez compte que la réutilisabilité vous facilite grandement la vie. Si la réutilisabilité n'existait pas, on programmerait encore en entrant des 0 et des 1 à la main dans l'ordinateur.

Créer une fonction ne demande pas beaucoup de travail. Pour voir comment on utilise des fonctions, ouvrez une copie d'IPython et tapez le code suivant (en appuyant sur Entrée à la fin de chaque ligne) :

```
def SayHello():  
    print('Hello les amis !')
```

Pour terminer la fonction, appuyez une seconde fois sur Entrée à la fin de la dernière ligne. Une fonction commence par le mot-clé `def` (pour *define*). Vous donnez à cette fonction un nom suivi de parenthèses pouvant contenir des *arguments* (données utilisées par la fonction) et d'un signe deux-points. L'éditeur effectue automatiquement un retrait sur la ligne suivante. Python utilise l'espace pour définir des blocs de code (parties de script associées l'une à l'autre dans une fonction).

La fonction peut alors être utilisée. Écrivez simplement **SayHello()** et appuyez sur Entrée. Les parenthèses qui suivent immédiatement le nom de la fonction sont importantes, car elles disent à Python qu'il doit exécuter la fonction plutôt que de vous signaler que vous accédez à une fonction en tant qu'objet (pour déterminer ce qu'est cet objet). Le résultat est l'affichage de l'expression Hello les amis !

Appeler des fonctions

Les fonctions peuvent accepter des arguments (qui sont des bits de données supplémentaires) et retourner des valeurs. Sans leur capacité d'échanger des données, les fonctions seraient considérablement moins utiles. Les sections qui suivent expliquent comment appeler des fonctions de diverses manières pour transmettre et recevoir des données.

Donner des arguments

Une fonction nécessite dans certains cas que le programme appelant lui transmette des arguments. Un argument requis est une variable qui doit obligatoirement contenir une donnée pour que la fonction puisse s'exécuter. Ouvrez une copie d'IPython et tapez le code suivant :

```
def DoSum(Value1, Value2):  
    return Value1 + Value2
```

Vous avez une nouvelle fonction, `DoSum()`. Cette fonction exige la fourniture de deux arguments. Du moins, c'est la notion que vous en aviez jusqu'à présent. Tapez **`DoSum()`** et appuyez sur Entrée. Vous obtenez un message d'erreur :

```
TypeError  
Traceback (most recent call last)  
<ipython-input-2-a37c1b30cd89> in <module>()  
----> 1 DoSum()  
TypeError: DoSum() missing 2 required positional  
arguments: 'Value1' and 'Value2'
```

Si vous essayez `DoSum()` avec un seul argument, vous obtiendrez un autre message d'erreur. Pour utiliser `DoSum()`, vous devez fournir deux arguments. Pour voir comment cela fonctionne, tapez **`DoSum(1, 2)`** et appuyez sur Entrée. Vous obtenez le résultat attendu : 3.



Il convient de remarquer que `DoSum()` produit une valeur de sortie de 3 quand on fournit en entrée 1 et 2. L'instruction `return` donne la valeur de l'output. Chaque fois que l'instruction `return` figure dans une fonction, cette fonction produit une valeur de sortie.

Transmettre des arguments par mot-clé

Quand vos fonctions deviennent plus complexes, de même que les méthodes pour les utiliser, il peut être souhaitable d'assurer un meilleur contrôle de la façon précise dont vous appelez la fonction et dont vous lui fournissez des arguments. Pour le moment, il s'agit d'arguments positionnels, ce qui signifie que les valeurs ont dû être fournies dans l'ordre dans lequel elles apparaissent dans la liste d'arguments de la définition de la fonction. Or, Python propose aussi une méthode pour transmettre les arguments par mot-clé. Il s'agit d'entrer le nom de l'argument suivi du signe égal (=) et de la valeur de l'argument. Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
def DisplaySum(Value1, Value2):  
    print(str(Value1) + ' + ' + str(Value2) + ' = ' +  
          str((Value1 + Value2)))
```



Il convient de remarquer que l'argument de l'instruction `print()` inclut une liste d'éléments à imprimer et que ces éléments sont séparés par des signes plus (+). En outre, les arguments sont de types différents, aussi devez-vous les convertir en utilisant la fonction `str()`. Python facilite ce mélange et cet assortiment des arguments. Cette fonction introduit aussi le concept de continuation automatique de la ligne. La fonction `print()` s'affiche sur deux lignes, et Python assure automatiquement la continuité de la fonction entre la première ligne et la seconde.

Ensuite, le moment est venu de tester `DisplaySum()`. Naturellement, il s'agit d'essayer la fonction en utilisant d'abord des arguments positionnels : tapez **`DisplaySum(2, 3)`** et appuyez sur Entrée. Le résultat s'affiche : `2 + 3 = 5`. Maintenant, tapez

DisplaySum(Value2 = 3, Value1 = 2) et appuyez sur Entrée. À nouveau, bien que les positions des arguments aient été inversées, le résultat s'affiche : $2 + 3 = 5$.

Affecter aux arguments de la fonction une valeur par défaut

Que l'appel des fonctions soit fait avec des arguments positionnels ou avec des arguments sous forme de mot-clé, il a fallu jusqu'ici leur fournir une valeur. Une fonction peut parfois utiliser des valeurs par défaut, lorsque l'on peut disposer d'une valeur courante. Les valeurs par défaut rendent la fonction plus facile à utiliser et moins susceptible de provoquer une erreur quand le développeur omet de fournir un input. Pour créer une valeur par défaut, faites simplement suivre le nom de l'argument par le signe égal et la valeur choisie. Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
def SayHello(Greeting = "Pas de valeur fournie"):  
    print(Greeting)
```

La fonction SayHello() fournit une valeur automatiquement pour Greeting quand le programme appelant n'en transmet pas. Si vous appelez SayHello() sans argument, cela n'entraîne pas un message d'erreur. Tapez **SayHello()** et appuyez sur Entrée pour le constater : vous verrez s'afficher le message par défaut. Tapez ensuite **SayHello("Salut ! ")** et vous obtiendrez une réponse normale.

Créer des fonctions avec un nombre variable d'arguments

Dans la plupart de cas, vous savez précisément combien d'arguments vous devez prévoir avec votre fonction. Il est avantageux de privilégier ce cas de figure, car lorsque les fonctions ont un nombre fixe d'arguments, il est plus facile de trouver les erreurs par la suite. Il arrive cependant que vous ne

puissiez pas déterminer le nombre d'arguments qu'une fonction recevra au départ. Ainsi, par exemple, si vous créez une application Python fonctionnant sur la ligne de commande, il se peut que l'utilisateur ne fournisse pas d'arguments, qu'il en fournisse le nombre maximum (en supposant qu'il en existe un), ou qu'il fournisse un nombre d'arguments quelconque dans cette fourchette.

Heureusement, Python offre une technique pour transmettre à une fonction un nombre variable d'arguments. Il suffit de créer un argument précédé d'un astérisque, comme `*VarArgs`. La technique usuelle consiste à fournir un second argument contenant le nombre d'arguments transmis en input. Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
def DisplayMulti(ArgCount = 0, *VarArgs):  
    print('Vous avez transmis ' + str(ArgCount) + '  
arguments.'  
        VarArgs)
```

Il convient de remarquer que la fonction `print()` affiche une chaîne puis la liste d'arguments. Compte tenu de la façon dont cette fonction est conçue, vous pouvez taper **DisplayMulti()** et appuyer sur Entrée pour constater qu'il vous est possible de transmettre zéro argument. Pour voir ce qui se produit quand vous transmettez plusieurs arguments, tapez **DisplayMulti(3, 'Hello', 1, True)** et appuyez sur Entrée. Le résultat de ('Vous avez transmis 3 arguments.', ('Hello', 1, True)) montre qu'il n'est pas nécessaire que les valeurs transmises soient d'un type particulier.

Utiliser des instructions conditionnelles et des boucles

Les algorithmes comportent souvent des décisions et des étapes qui doivent être répétées. Par exemple, si l'on doit filtrer des données, il se peut qu'une décision soit nécessaire pour rejeter éventuellement une valeur qui ne serait pas conforme, ou bien il

peut être nécessaire de traiter des données plus d'une fois pour obtenir le résultat désiré. Python prévoit des instructions spéciales pour les décisions et pour les répétitions, comme on va le voir dans les sections suivantes.

Prendre des décisions en utilisant l'instruction if

Avec des « si » (if), vous envisagez des conditions à tout moment dans votre quotidien. Vous vous dites par exemple : « Si le temps le permet, nous irons pique-niquer dimanche. » L'instruction if dans Python est un peu verbeuse, mais le principe est le même. Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
def TestValue(Value):
    if Value == 5:
        print('La valeur est 5!')
    elif Value == 6:
        print('La valeur est 6!')
    else:
        print('C'est une autre valeur.')
        print('It equals ' + str(Value))
```

Une instruction if est une instruction qui commence par le mot *if*, qui indique à Python que vous voulez une décision. Le mot *if* est suivi d'une condition. La *condition* énonce la comparaison à laquelle Python doit procéder. En l'occurrence, Python doit déterminer si Value contient la valeur 5.



Il convient de remarquer que la condition utilise l'opérateur relationnel d'égalité `==`, et non pas l'opérateur d'affectation `=`. Une erreur courante que font les développeurs est d'utiliser l'opérateur d'affectation au lieu de l'opérateur d'égalité, ce qui ne peut qu'entraîner un dysfonctionnement du code.

Une condition se termine toujours par des deux-points (`:`). Si vous avez oublié ce symbole, Python ne saura pas que la condition

est terminée et continuera de chercher des conditions supplémentaires sur lesquelles la décision doit se fonder. Les deux-points sont suivis des tâches que Python doit exécuter.

Il arrive souvent qu'une instruction corresponde à plusieurs tâches à exécuter. La clause `elif` permet d'ajouter une condition supplémentaire et des tâches associées. Une clause est un ajout à une condition, laquelle, en l'occurrence, est une instruction en `if`. La clause `elif` introduit toujours une condition, tout comme l'instruction en `if`, et elle s'accompagne de sa propre série de tâches à exécuter.

Parfois, une instruction doit être exécutée quelle que soit la condition. Dans ce cas, on ajoute la clause `else`. La clause `else` indique à Python qu'il doit exécuter une certaine tâche quand les conditions de l'instruction en `if` ne sont pas remplies.



On peut noter que l'indentation prend davantage d'importance quand les fonctions deviennent plus complexes. La fonction comporte une instruction en `if`. L'instruction en `if` comporte une seule instruction `print()`. La clause `else` comporte deux instructions `print()`.

Pour tester cette fonction, tapez **TestValue(1)** et appuyez sur Entrée. Le résultat de la clause `else` apparaît. Tapez **TestValue(5)** et appuyez sur Entrée. L'output reflète à présent l'instruction `if`. Tapez **TestValue(6)** et appuyez sur Entrée. L'output est alors le résultat de la clause `elif`. La fonction est ainsi plus flexible que les fonctions déjà étudiées dans ce chapitre, car elle peut prendre des décisions.

Choisir entre plusieurs possibilités grâce à des décisions imbriquées

L'*imbrication* consiste à placer une instruction subordonnée à l'intérieur d'une autre instruction. Dans la plupart des cas, toute instruction peut être imbriquée dans une autre. Pour voir comment

cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
def SecretNumber():
    One = int(input("Entrez un nombre entre 1 et 10:
"))
    Two = int(input("Entrez un nombre entre 1 et 10:
"))

    if (One >= 1) and (One <= 10):
        if (Two >= 1) and (Two <= 10):
            print('Votre nombre secret est : ' + str(One
* Two))
        else:
            print("Seconde valeur incorrecte !")
    else:
        print("Première valeur incorrecte !")
```

Ici, SecretNumber() vous demande de fournir deux inputs. En effet, vous pouvez faire en sorte qu'un programme demande à l'utilisateur de saisir des entrées en cas de besoin, au moyen de la fonction input(). La fonction input() convertit les entrées en nombres.

Cette fois, nous avons deux niveaux d'instructions. Le premier niveau vérifie la validité du nombre saisi dans la variable One. Le second niveau vérifie la validité du nombre saisi dans la variable Two. Si les valeurs de One et Two sont toutes deux comprises entre 1 et 10, SecretNumber() donne à l'utilisateur un nombre secret.

Pour un exemple de fonctionnement de SecretNumber(), tapez **SecretNumber()** et appuyez sur Entrée. Le programme vous demande un premier input : tapez **20** et appuyez sur Entrée ; le programme vous en demande un second, tapez **10** et appuyez sur Entrée. Un message d'erreur vous signale que la première valeur est incorrecte. À nouveau, tapez **SecretNumber()** et appuyez sur Entrée. Cette fois, saisissez les valeurs 10 et 20. La fonction vous indique que le second input est incorrect. Essayez la même séquence avec les valeurs 10 et 10.

Exécuter des tâches répétitives en utilisant une boucle for

Il est parfois nécessaire qu'une tâche soit exécutée plusieurs fois de suite. Pour qu'une tâche soit exécutée un nombre donné de fois, on utilise la boucle for. La boucle for a un début et une fin, qui doivent être bien définis. Le nombre d'exécutions assurées par la boucle dépend du nombre d'éléments dans la variable. Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
def DisplayMulti(*VarArgs):
    for Arg in VarArgs:
        if Arg.upper() == 'CONT':
            continue

            print('Continue Argument: ' + Arg)
        elif Arg.upper() == 'BREAK':
            break

            print('Break Argument: ' + Arg)
        print('Good Argument: ' + Arg)
```

Ici, la boucle for commence à traiter un par un les éléments de VarArgs. Remarquez qu'une instruction est imbriquée dans la boucle et qu'elle teste deux conditions d'arrêt. Dans la plupart des cas, le code saute l'instruction if et affiche simplement l'argument. Cependant, quand l'instruction if rencontre le mot CONT ou le mot BREAK dans les valeurs en input, elle exécute une de ces deux tâches :

- » continue : Oblige la boucle à continuer l'exécution au-delà de ce point, avec la prochaine entrée dans VarArgs.
- » break : Met fin à l'exécution de la boucle.



Les mots-clés peuvent apparaître en utilisant une combinaison de majuscules et minuscules, comme ConT, sachant que la fonction upper() les convertit en majuscules. La fonction DisplayMulti() peut traiter un nombre quelconque de chaînes en input. Pour le

constater, tapez **DisplayMulti('Bonjour', 'Au revoir', 'Premier', 'Dernier')** et appuyez sur Entrée. Chacune des chaînes entrées apparaît sur une nouvelle ligne en output. À présent, tapez **DisplayMulti('Hello', 'Cont', 'Au revoir', 'Break', 'Last')** et appuyez sur Entrée. Vous pouvez constater que les chaînes Cont et Break n'apparaissent pas en output. C'est parce que ce sont des mots-clés. En outre, le mot Last n'apparaît pas en output, car la boucle for s'arrête avant de traiter ce mot.

Utiliser l'instruction while

L'instruction de boucle while continue à exécuter les tâches jusqu'à ce que la condition ne soit plus vraie. Comme la boucle for, la boucle while accepte les mots-clés continue et break comme instructions d'arrêt prématuré. Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
def SecretNumber():
    GotIt = False
    while GotIt == False:
        One = int(input("Saisissez un nombre entre 1 et
10: "))
        Two = int(input("Saisissez un nombre entre 1 et
10: "))

        if (One >= 1) and (One <= 10):
            if (Two >= 1) and (Two <= 10):
                print('Le nombre secret est : ' + str(One *
Two))
                GotIt = True
                continue
            else:
                print("Seconde valeur incorrecte !")
        else:
            print("Première valeur incorrecte !")
            print("Essayez encore !")
```

Il s'agit d'une forme plus développée de la fonction SecretNumber() présentée initialement dans la section « Choisir entre plusieurs possibilités grâce à des décisions imbriquées », précédemment dans ce chapitre. Ici, cependant, on ajoute

l'instruction de boucle `while` pour que la fonction continue à demander à l'utilisateur un input jusqu'à ce qu'elle reçoive une réponse valide.

Pour voir comment fonctionne une boucle `while`, tapez **SecretNumber()** et appuyez sur Entrée. À la première invite, tapez **20** et appuyez sur Entrée. À la seconde invite, tapez **10** et appuyez sur Entrée. Dans cet exemple, le programme vous signale que le premier nombre est incorrect et vous invite à réessayer. Faites une deuxième tentative en saisissant successivement les valeurs 10 et 20. Cette fois, c'est la seconde valeur qui est incorrecte et vous êtes invité à réessayer encore. Pour votre troisième essai, saisissez 10 et 10. Cette fois, vous obtenez le nombre secret. Vous remarquerez qu'en raison de la clause `continue`, l'application ne vous demande plus de réessayer.

Stocker des données à l'aide d'ensembles, de listes et de tuples

L'algorithmique est avant tout affaire de données. Python offre tout un ensemble de méthodes pour stocker les données en mémoire. Chaque méthode a ses avantages et ses inconvénients. Pour chaque besoin particulier, il est important de choisir la méthode la plus appropriée. Les sections suivantes présentent trois techniques couramment utilisées dans le domaine de la science des données pour stocker l'information.

Créer des ensembles

Pour la plupart d'entre nous, nous avons déjà manié des ensembles à un moment ou à un autre de notre scolarité, pour créer des listes d'éléments qui vont ensemble. Ensuite, ces listes ont fait l'objet de manipulations à l'aide d'opérations mathématiques comme l'intersection, l'union, la différence et la différence symétrique. Les ensembles sont le meilleur choix à

faire quand il s'agit de procéder à des tests d'appartenance ou de supprimer des doublons dans une liste. En revanche, l'utilisation des ensembles ne permet pas d'exécuter des tâches séquentielles comme l'indexation ou le découpage. Pour un exemple d'utilisation des ensembles, lancez une copie d'IPython et tapez le code suivant :

```
SetA = set(['Rouge', 'Bleu', 'Vert', 'Noir'])
SetB = set(['Noir', 'Vert', 'Jaune', 'Orange'])
SetX = SetA.union(SetB)
SetY = SetA.intersection(SetB)
SetZ = SetA.difference(SetB)
```

Vous disposez ainsi de cinq ensembles différents, avec des éléments communs. Pour voir les résultats de chaque opération mathématique, tapez **print('{0}\n{1}\n{2}'.format(SetX, SetY, SetZ))** et appuyez sur Entrée. Chaque ligne du résultat montre le contenu d'un ensemble :

```
{'Bleu', 'Orange', 'Rouge', 'Vert', 'Noir', 'Jaune'}
{'Vert', 'Noir'}
{'Bleu', 'Rouge'}
```



Vous obtenez les résultats des opérations mathématiques : union(), intersection() et difference(). Le format raffiné de l'impression de Python peut être appréciable quand on travaille avec des collections comme les ensembles. La fonction format() indique à Python quels objets doivent être placés dans chacun des espaces réservés de la chaîne. Un *espace réservé* est une série de paires d'accolades ({}), pouvant chacune contenir éventuellement un nombre. Le *caractère d'échappement* \n crée une nouvelle ligne entre les entrées (les caractères d'échappement sont des caractères de contrôle spéciaux). Pour plus de détails sur l'amélioration du format de présentation, consultez la page <https://docs.python.org/3/tutorial/inputoutput.html>.

Vous pouvez aussi tester les relations entre ces différents ensembles. Par exemple, tapez **SetA.issuperset(SetY)** et appuyez sur Entrée. La valeur de sortie True vous indique que SetA contient SetY. De même, si vous tapez **SetA.issubset(SetX)** et

appuyez sur Entrée, le résultat vous indique que SetA est un sous-ensemble de SetX.

Il est important de noter que les ensembles sont soit *mutables*, c'est-à-dire modifiables, soit *immutables*. Tous les ensembles de cet exemple sont mutables, ce qui signifie qu'on peut y ajouter des éléments ou en retirer. Ainsi, si vous tapez **SetA.add('Pourpre')** et appuyez sur Entrée, SetA recevra un nouvel élément. Ensuite, si vous tapez **SetA.issubset(SetX)** et appuyez sur Entrée, vous constaterez que SetA n'est plus un sous-ensemble de SetX, car il contient maintenant l'élément 'Pourpre'.

Créer des listes

Dans les spécifications de Python, une liste est un type de séquence. Les *séquences* permettent de rassembler plusieurs éléments de données dans une même unité de stockage, mais en tant qu'entités distinctes. On peut comparer cela aux blocs de boîtes à lettres dans les immeubles des grandes copropriétés. Chaque bloc est constitué d'un certain nombre de boîtes individuelles, et chaque boîte peut contenir du courrier. Python permet d'utiliser également d'autres types de séquences :

- » **Les tuples** : Un tuple est une collection utilisée pour créer des séquences complexes sur le modèle des listes. Un avantage des tuples est la possibilité d'en imbriquer le contenu. Cette propriété vous permet de créer des structures pour enregistrer aussi bien les dossiers de vos salariés que des paires x-y coordonnées.
- » **Les dictionnaires** : À l'image des vrais dictionnaires, vous créez des paires clé/valeur (comme un mot et la définition qui lui est associée). Un dictionnaire permet d'effectuer des recherches avec une rapidité considérable et de faciliter significativement la mise en ordre des données.
- » **Les piles** : La plupart des langages de programmation supportent directement les piles. Ce n'est cependant pas le

cas de Python, mais il existe une solution de rechange. Une pile est une séquence de type LIFO (*last in/first out*, c'est-à-dire dernier entré, premier sorti). Imaginons une pile de pièces de monnaie : vous pouvez ajouter des pièces en haut de la pile, et vous pouvez retirer des pièces du haut de la pile. Ici, une pile est une collection importante que l'on peut simuler au moyen d'une liste.

- » **Les files** : Une file est une collection de type FIFO (*first in/first out*, c'est-à-dire premier entré, premier sorti). Elle vous sert à suivre des éléments qui doivent être traités d'une certaine façon. C'est un peu comme une file d'attente devant les guichets d'une banque. Vous entrez dans la file, vous attendez votre tour, et finalement un guichetier vous appelle.
- » **Les deque** : Une queue à double fin (appelée un *deque*) est une structure de file dans laquelle vous pouvez ajouter ou retirer des éléments aux deux extrémités, mais pas au milieu. Vous pouvez utiliser un deque comme une file ou comme une pile, ou comme n'importe quel type de collection à laquelle vous ajoutez et retirez des éléments de façon ordonnée (par opposition aux listes, aux tuples et aux dictionnaires, qui permettent un accès et une gestion aléatoires).

De toutes les séquences, les listes sont le type le plus simple et le plus directement lié aux objets du monde réel. Le travail avec des listes vous rend plus capable d'utiliser d'autres types de séquences qui offrent davantage de fonctionnalités et de flexibilité. Les données sont stockées dans une liste comme on note une liste sur une feuille de papier : un élément arrive après un autre. Une liste a un début, un milieu et une fin. Python numérote les éléments d'une liste (même si vous n'avez pas l'habitude de numérotter les éléments des listes que vous dressez dans la pratique, la numérotation facilite l'accès aux éléments). Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
ListA = [0, 1, 2, 3]
ListB = [4, 5, 6, 7]
ListA.extend(ListB)
ListA
```

Quand vous avez tapé la dernière ligne de code, vous voyez le résultat de [0, 1, 2, 3, 4, 5,

6, 7]. La fonction `extend()` ajoute les membres de `ListB` à `ListA`. Vous pouvez également le faire à l'aide de la fonction `append()`. Tapez **ListA.append(-5)** et appuyez sur Entrée. Quand vous tapez **ListA** et appuyez sur Entrée, vous constatez que Python a ajouté - 5 à la fin de la liste. Si vous voulez maintenant supprimer des éléments, vous pouvez le faire en utilisant la fonction `remove()`. Par exemple, tapez **ListA.remove(-5)** et appuyez sur Entrée. Ensuite, contrôlez à nouveau le contenu de `ListA` en tapant **ListA** et en appuyant sur Entrée, et vous constaterez que l'entrée ajoutée n'est plus là.



Les listes se prêtent aussi à la *concaténation*. Pour ajouter une liste à une autre, on utilise le signe plus (+). Ainsi, par exemple, si vous tapez **ListX = ListA + ListB** et appuyez sur Entrée, vous constaterez que la liste `ListX` nouvellement créée contient `ListA` et `ListB`, les éléments de `ListA` venant en premier.

Créer et utiliser des tuples

Un tuple est une collection utilisée pour créer des listes complexes. Il est possible d'imbriquer un tuple dans un autre. Par conséquent, vous pouvez créer des hiérarchies entre les tuples. Une hiérarchie peut être quelque chose d'aussi simple que l'arborescence des répertoires de votre disque dur ou l'organigramme de votre entreprise. Surtout, il est possible de créer des structures de données complexes en utilisant un tuple.



Les tuples ne sont pas mutables. Vous pouvez créer un nouveau tuple en lui donnant le même nom et le modifier, mais vous ne pouvez pas modifier un tuple existant. Les listes sont mutables,

donc modifiables. On peut donc penser que les tuples présentent ainsi un inconvénient, or l'immuabilité présente toutes sortes d'avantages : notamment, plus de sécurité et plus de rapidité. En outre, les objets immutables sont plus faciles à traiter par des systèmes multiprocesseurs. Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
MonTuple = (1, 2, 3, (4, 5, 6, (7, 8, 9)))
```

MonTuple présente trois niveaux d'imbrication. Le premier niveau est constitué des valeurs 1, 2, 3 et d'un tuple. Le deuxième niveau est constitué des valeurs 4, 5, 6 et d'un autre tuple. Le troisième niveau est constitué des valeurs 7, 8, et 9. Pour voir comment cela fonctionne, ouvrez une copie d'IPython et tapez le code suivant :

```
for Value1 in MyTuple:
    if type(Value1) == int:
        print(Value1)
    else:
        for Value2 in Value1:
            if type(Value2) == int:
                print("\t", Value2)
            else:
                for Value3 in Value2:
                    print("\t\t", Value3)
```

Quand vous exécutez ce code, vous constatez que les valeurs se situent réellement à trois niveaux différents. Les indentations indiquent les niveaux :

```
1
2
3
    4
    5
    6
        7
        8
        9
```



Il est possible d'exécuter des tâches comme ajouter de nouvelles valeurs, mais il faut pour cela ajouter les entrées initiales et les nouvelles valeurs à un nouveau tuple. De plus, vous ne pouvez ajouter des tuples qu'à un tuple existant. Pour voir comment cela fonctionne, tapez **MyNewTuple = MyTuple.__add__((10, 11, 12, (13, 14, 15)))** et appuyez sur Entrée. MyNewTuple contient les nouvelles entrées au premier et au deuxième niveaux : (1, 2, 3, (4, 5, 6, (7, 8, 9)), 10, 11, 12, (13, 14, 15)).

Définir des itérateurs utiles

Les chapitres qui suivent utilisent toutes sortes de techniques pour accéder à des valeurs une par une dans divers types de structures de données. Pour cette section, on utilise deux listes simples définies comme suit :

```
ListA = ['Orange', 'Jaune', 'Vert', 'Marron']  
ListB = [1, 2, 3, 4]
```

La méthode la plus simple pour accéder à une valeur consiste à utiliser un index. Ainsi, par exemple, si vous tapez **ListA[1]** et appuyez sur Entrée, vous verrez s'afficher comme résultat 'Jaune'. Dans Python, tous les index sont basés sur zéro, ce qui signifie que la première entrée est 0, et non pas 1.

Les rangs constituent une autre méthode simple pour accéder aux valeurs. Ainsi, par exemple, si vous tapez **ListB[1 : 3]** et appuyez sur Entrée, le résultat sera [2, 3]. Vous pouvez utiliser le rang comme input d'une boucle for :

```
for Value in ListB[1:3]:  
    print(Value)
```

Au lieu de la liste entière, on ne voit apparaître que 2 et 3 comme outputs, sur deux lignes distinctes. Le rang comporte deux valeurs séparées par un deux-points. Cependant, les valeurs sont facultatives. Ainsi, par exemple, **ListB[: 3]** donne [1, 2, 3].

Quand on omet une valeur, le rang commence au début ou à la fin de la liste, selon le cas.

Il arrive que l'on doive traiter deux listes en parallèle. Pour ce faire, la méthode la plus simple consiste à utiliser la fonction `zip()`. Voici un exemple d'utilisation de cette fonction :

```
for Value1, Value2 in zip(ListA, ListB):  
    print(Value1, '\t', Value2)
```

Ce code traite `ListA` et `ListB` en même temps. Le traitement est terminé quand la boucle `for` traite la liste la plus courte. En l'occurrence, le résultat est le suivant :

```
Orange 1  
Jaune  2  
Vert   3  
Marron 4
```



C'est là le sommet de l'iceberg. Dans ce livre, vous rencontrerez toute une série de types d'itérateurs. Le principe est de pouvoir lister seulement les éléments qui vous intéressent, plutôt que tous les éléments d'une liste ou autre structure de données. Certains des itérateurs utilisés dans les chapitres suivants sont un peu plus compliqués que ce que vous pouvez voir ici, mais il s'agit d'un point de départ important.

Indexer les données à l'aide de dictionnaires

Un dictionnaire est une séquence d'un type particulier qui utilise des paires d'éléments constituées d'un nom et d'une valeur. L'utilisation d'un nom facilite l'accès à des valeurs avec autre chose qu'un index numérique. Pour créer un dictionnaire, on place entre accolades des paires (nom, valeur). Créez un dictionnaire en tapant **MonDico = {'Orange' : 1, 'Bleu' : 2, 'Rose' : 3}** et en appuyant sur Entrée.

Pour accéder à une valeur particulière, utilisez le nom et un index. Par exemple, tapez **MonDico['Rose']** et appuyez sur Entrée. Le résultat est la valeur 3. L'utilisation de dictionnaires comme structures de données facilite l'accès à des jeux de données extrêmement complexes à l'aide de termes que tout le monde peut comprendre. Par bien d'autres aspects, l'utilisation d'un dictionnaire ne diffère pas de l'utilisation d'un autre type de séquence.

Les dictionnaires présentent cependant des caractéristiques qui leur sont propres. Tapez **MonDico.keys()** et appuyez sur Entrée pour faire apparaître une liste des clés. Vous pouvez utiliser la fonction **values()** pour voir la liste des valeurs qui se trouvent dans le dictionnaire.

Chapitre 5

Effectuer des manipulations de données essentielles à l'aide de Python

DANS CE CHAPITRE

- » Utiliser des matrices et des vecteurs pour effectuer des calculs
 - » Obtenir les combinaisons correctes
 - » Recourir à des techniques récursives pour obtenir les résultats désirés
 - » Étudier des moyens d'accélérer les calculs
-

Le [Chapitre 4](#) traite de l'utilisation de Python comme moyen d'exprimer en termes concrets ces symboles mystérieux que l'on retrouve souvent dans les représentations mathématiques des algorithmes. Dans ce chapitre, vous allez découvrir les diverses constructions de langage utilisées dans Python pour effectuer des tâches. Cependant, il ne suffit pas de simplement savoir maîtriser un langage en utilisant ses constructions pour effectuer des tâches. Le but des algorithmes mathématiques est de transformer un type de données en un autre type de données. *Manipuler des données* signifie effectuer une transformation à partir d'intrants bruts afin d'obtenir un résultat voulu (à l'instar de la science des données, ce sujet est traité dans *Python for Data Science For Dummies*, par John Paul Mueller et Luca Massaron, éditions Wiley). Les données relatives à la circulation automobile,

par exemple, ne vous apportent aucune information tant qu'elles se présentent dans leur forme brute : vous devez les manipuler pour pouvoir constater des tendances et comprendre quelles dépenses d'amélioration vous pouvez envisager. Ces symboles quelque peu ésotériques ont donc leur utilité. Vous les utilisez comme une sorte de machine pour transformer les données brutes en quelque chose d'exploitable, comme vous allez le découvrir dans ce chapitre.

Dans le passé, il fallait effectuer à la main les différentes manipulations nécessaires pour rendre les données exploitables, et cela supposait des connaissances poussées en mathématiques. Heureusement, les modules de Python vous permettent aujourd'hui d'effectuer la plupart de ces manipulations au moyen de quelques lignes de code. Vous n'êtes plus obligé de mémoriser des manipulations compliquées : il vous suffit de savoir quelles fonctionnalités de Python vous devez utiliser. C'est cette compétence que vous allez acquérir grâce à ce chapitre. Vous allez découvrir comment effectuer divers types de manipulations de données en utilisant des modules Python faciles d'accès et conçus spécialement pour cet usage. Ce chapitre commence avec des manipulations de vecteurs et de matrices. Les sections qui suivent étudient des techniques comme la récursivité, qui rendent les tâches encore plus simples et permettent l'exécution de certaines tâches qui seraient pratiquement impossibles en utilisant d'autres moyens. Vous allez découvrir aussi comment accélérer les calculs en vue de pouvoir consacrer moins de temps à manipuler les données et plus de temps à en faire quelque chose de vraiment intéressant, comme apprendre à éviter que tous ces bouchons de circulation se produisent.

Effectuer des calculs avec des vecteurs et des matrices

Souvent, pour pouvoir travailler efficacement avec Python, vous devez manipuler d'importantes quantités de données qui se

présentent sous des formats particuliers. Ces formats ont des noms, qui peuvent vous sembler un peu rebutants, mais qui sont importants. Dans ce chapitre, il y a trois termes que vous devez connaître :

- » **Scalaire** : Élément de donnée unique. Le nombre 2, par exemple, est un scalaire.
- » **Vecteur** : Objet unidimensionnel (assimilable à une liste) constitué d'éléments de données. Le quadruplet constitué des nombres 2, 3, 4 et 5, par exemple, peut être un vecteur. On accède aux éléments (ou coordonnées) d'un vecteur en utilisant un *indice* de base 0, ou si vous préférez, un pointeur vers l'élément voulu. L'élément dont l'indice est 0 est le premier élément du vecteur, en l'occurrence 2.
- » **Matrice** : Objet à deux ou plusieurs dimensions (assimilable à un tableau) constitué d'éléments de données. Une matrice à deux dimensions peut être constituée, par exemple, des nombres 2, 3, 4 et 5 dans sa première rangée (ou ligne) et des nombres 6, 7, 8 et 9 dans sa seconde rangée. On accède aux éléments d'une matrice en utilisant un double indice ligne-colonne de base 0. L'élément de la ligne 0 et de la colonne 0 est le premier élément de la matrice, en l'occurrence, 2.

Python propose un intéressant assortiment de fonctionnalités propres, décrites au [Chapitre 4](#), mais avec lesquelles certaines tâches demandent encore beaucoup de travail. Pour réduire cette quantité de travail nécessaire, vous pouvez vous servir d'un code écrit par d'autres, et conditionné sous forme de modules. Les sections suivantes expliquent comment utiliser le module NumPy pour exécuter des tâches variées sur des scalaires, des vecteurs et des matrices.

Maîtriser les opérations sur les scalaires et les vecteurs

Le module NumPy apporte une fonctionnalité essentielle pour les calculs scientifiques dans Python. Pour pouvoir l'utiliser, importez-le en tapant une commande comme `import numpy as np`. Vous pouvez alors accéder à numpy en utilisant l'abréviation courante de deux lettres `np`.



Python donne accès à un seul type de donnée dans une catégorie particulière. Ainsi, pour créer une variable qui représente un nombre sans partie décimale, vous utilisez le type entier. Une telle désignation générique est utile, car elle simplifie le code et délivre le développeur de bien des soucis. Cependant, dans les calculs scientifiques, on a souvent besoin de mieux contrôler la façon dont les données apparaissent en mémoire, ce qui implique de disposer de davantage de types de données, ce qui est possible grâce à numpy. Supposons, par exemple, que vous ayez besoin de définir un scalaire particulier comme `short` (valeur codée sur 16 bits). Avec numpy, vous pouvez le définir ainsi : `myShort = np.short(15)`. Vous pouvez définir une variable de la même taille en utilisant la fonction `np.int16`. Le module NumPy vous donne accès à un assortiment secondaire de types de données, voir <https://docs.scipy.org/doc/numpy/reference/arrays.scalars.html>.

Utilisez la fonction numpy `array` pour créer un vecteur. Ainsi, par exemple, `myVect = np.array([1, 2, 3, 4])` crée un vecteur de quatre éléments. Ici, le vecteur est constitué d'entiers de type standard de Python. Vous pouvez aussi utiliser la fonction `arange` pour produire des vecteurs, comme `myVect = np.arange(1, 10, 2)` qui affecte à `myVect` le contenu `array([1, 3, 5, 7, 9])`. La première entrée indique le point de départ, la deuxième le point d'arrêt, et la troisième le pas entre deux nombres. Un quatrième argument vous permet de définir le type de données pour le vecteur. Vous pouvez également créer un vecteur avec un type de données spécifique. Il vous suffit de spécifier le type de données, comme par exemple `myVect = np.array(np.int16([1, 2, 3, 4]))`, pour produire un vecteur `myVect` se présentant comme suit : `array([1, 2, 3, 4], dtype=int16)`.

Dans certains cas, il vous faut des fonctions numpy spéciales pour créer un vecteur (ou une matrice) d'un type particulier. Pour

certaines tâches mathématiques, par exemple, il est nécessaire de remplir le vecteur avec des « 1 ». Vous utiliserez alors la fonction `ones` comme ceci, `myVect = np.ones(4, dtype=np.int16)`, pour remplir `myVect` avec des « 1 » d'un type de donnée spécifique, comme ceci : `array([1, 1, 1, 1], dtype=int16)`. Vous pouvez aussi utiliser la fonction `zeros` pour remplir de zéros un vecteur.



Vous pouvez appliquer des fonctions mathématiques de base sur des vecteurs de façon globale, ce qui est extrêmement utile et réduit le risque d'erreurs dans le contexte de la programmation de constructions comme les boucles, pour l'exécution d'une même tâche. Ainsi, par exemple, avec des entiers de type standard de Python, `myVect + 1` produit comme output `array([2, 3, 4, 5])`, alors que si vous utilisez le type de données numpy `int16`, `myVect + 1` produit comme output `array([2, 3, 4, 5], dtype=int16)`. Il convient de noter que l'output vous indique quel type de données est utilisé. Comme on pouvait s'y attendre, `myVect - 1` produit comme output `array([0, 1, 2, 3])`. Vous pouvez même utiliser des vecteurs dans des exemples mathématiques plus complexes, comme `2 ** myVect`, où l'output est `array([2, 4, 8, 16], dtype=int32)`. Cependant, utilisé de cette manière, numpy assigne souvent un type spécifique à l'output, même lorsque vous définissez un vecteur en utilisant des entiers de type standard de Python.

Enfin, concernant les opérations sur les scalaires et les vecteurs, vous pouvez aussi effectuer des tâches logiques et de comparaison. Ainsi, par exemple, le code qui suit exécute des opérations de comparaison entre deux tableaux :

```
a = np.array([1, 2, 3, 4])
b = np.array([2, 2, 4, 4])

a == b
array([False, True, False, True], dtype=bool)
a < b
array([ True, False, True, False], dtype=bool)
```

À partir de deux vecteurs `a` et `b`, le code vérifie si les éléments de `a` sont égaux à ceux de `b`. Dans notre exemple, `a[0]` n'est pas égal

à `b[0]` mais `a[1]` est bien égal à `b[1]`. L'output est un vecteur de type `bool` contenant les valeurs `True` (Vrai) ou `False` (Faux) selon le résultat des comparaisons. De même, vous pouvez tester dans quels cas `a < b` et créer un autre vecteur contenant les valeurs de vérité correspondant à cet exemple.

Les opérations logiques se font à l'aide de fonctions spéciales. On vérifie le résultat logique des opérateurs booléens AND, OR, XOR et NOT. Voici un exemple de fonctions logiques :

```
a = np.array([True, False, True, False])
b = np.array([True, True, False, False])

np.logical_or(a, b)
array([ True,  True,  True, False], dtype=bool)

np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)

np.logical_not(a)
array([False,  True, False,  True], dtype=bool)

np.logical_xor(a, b)
array([False,  True,  True, False], dtype=bool)
```

Vous pouvez également utiliser ces fonctions avec un input numérique. Dans ce cas, 0 signifie Faux et 1 signifie Vrai. Comme pour les comparaisons, les fonctions procèdent élément par élément, même si vous faites un seul appel. Pour plus de détails sur les fonctions logiques, consultez la page <https://docs.scipy.org/doc/numpy-1.10.0/reference/routines.logic.html>.

Effectuer une multiplication vectorielle

L'addition, la soustraction et la division sur les vecteurs se font élément par élément, comme indiqué dans la section précédente. En ce qui concerne la multiplication, les choses sont un peu plus

compliquées. Elles peuvent même devenir vraiment compliquées, tout dépend de ce que vous voulez faire. Considérons le type de multiplication dont il a été question dans la section précédente. Les deux formules `myVect * myVect` et `np.multiply(myVect, myVect)` produisent l'output `array([1, 4, 9, 16])`, calculé élément par élément.



Malheureusement, une multiplication élément par élément peut produire des résultats incorrects quand on travaille avec des algorithmes. Bien souvent, ce dont on a réellement besoin, c'est du *produit scalaire*, qui est la somme des produits de deux séries de nombres. Avec les vecteurs, le produit scalaire est toujours la somme de multiplications effectuées élément par élément, et le résultat est un nombre unique. Ainsi, par exemple, `myVect.dot(myVect)` donne comme output 30. Quand on additionne les valeurs sur lesquelles est effectuée la multiplication élément par élément, on trouve effectivement 30. La page <https://www.mathsisfun.com/algebra/vectors-dot-product.html> vous en dit davantage sur les produits scalaires et vous explique dans quels cas ils sont appropriés en algorithmique. Pour en savoir davantage sur les fonctions de manipulation en algèbre linéaire pour numpy, consultez la page <https://docs.scipy.org/doc/numpy/reference/routines.linalg.html>.

Créer une matrice est la bonne façon de commencer

Les techniques utilisées avec les vecteurs sont souvent applicables aux matrices. Pour créer une matrice de base, il vous suffit d'utiliser la fonction `array` comme vous le feriez avec un vecteur, mais en définissant des dimensions supplémentaires. Une dimension est une direction au sein de la matrice. Une matrice bidimensionnelle, par exemple, comporte deux ou plusieurs lignes (une direction) et deux ou plusieurs colonnes (seconde direction). L'appel `myMatrix = np.array([[1,2,3], [4,5,6], [7,8,9]])` produit une matrice comportant trois lignes et trois colonnes :

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Ici, on imbrique trois listes dans une compréhension de listes pour créer les deux dimensions. Pour accéder à un élément particulier, on fournit une valeur d'indice pour la ligne et pour la colonne, par exemple `myMatrix[0, 0]` pour accéder à la première valeur, 1. On peut produire des matrices de dimension quelconque en utilisant une technique similaire. Par exemple, `myMatrix = np.array([[[1,2], [3,4]], [[5,6], [7,8]]])` produit une matrice tridimensionnelle, c'est-à-dire avec trois axes x, y et z, qui se présente ainsi :

```
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
```

Dans cet exemple, on imbrique deux listes dans deux compréhensions de listes, le tout dans une unique compréhension de listes qui englobe l'ensemble. Ici, il faut fournir une valeur d'indice x, y, z pour accéder à une valeur particulière. Par exemple, `myMatrix[0, 1, 1]` accède à la valeur 4.



Dans certains cas, on a besoin de créer une matrice comportant certaines valeurs de départ. Si vous avez besoin d'une matrice qui soit initialement remplie de « 1 », par exemple, vous pouvez utiliser la fonction `ones`. L'appel de `myMatrix = np.ones([4,4], dtype=np.int32)` produit une matrice de quatre lignes et quatre colonnes remplie de valeurs `int32` :

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

De même, l'appel de `myMatrix = np.ones([4,4,4], dtype=np.bool)` engendre un tableau tridimensionnel. Cette fois, la matrice contiendra les valeurs booléennes `True`. Il existe aussi des

fonctions pour créer une matrice remplie de zéros, la matrice identité, ou pour répondre à d'autres besoins. Vous trouverez une liste complète de fonctions de création de vecteurs et de matrices à l'adresse

<https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html>.



Le module NumPy supporte une véritable classe matrix. La classe matrix supporte des spécificités qui facilitent l'exécution des tâches spécifiques aux matrices. Vous découvrirez ces spécificités plus loin dans ce chapitre. Pour l'instant, vous avez simplement besoin de savoir comment créer une matrice du type de données matrix. La méthode la plus facile consiste à lancer un appel de fonction similaire à celui que vous utilisez pour la fonction array, mais en utilisant ici la fonction mat, comme dans l'exemple `myMatrix = np.mat([[1,2,3], [4,5,6], [7,8,9]])` qui donne la matrice suivante :

```
matrix([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

Vous pouvez aussi convertir une collection existante en matrice à l'aide de la fonction asmatrix. Utilisez la fonction asarray pour remettre un objet matrix sous la forme array.



Le seul problème avec la classe matrix est qu'elle n'est utilisable que pour les matrices à deux dimensions. Si vous tentez de convertir une matrice tridimensionnelle en classe matrix, vous obtiendrez un message d'erreur qui vous indiquera que de par sa dimension, cette forme ne peut pas être une matrice.

Multiplier les matrices

La multiplication de deux matrices pose les mêmes problèmes que la multiplication de deux vecteurs (voir la section « Effectuer une multiplication vectorielle », précédemment dans ce chapitre). Le code suivant exécute une multiplication de deux matrices, qui s'effectue élément par élément.

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[1,2,3],[4,5,6]])

a*b
array([[ 1,  4,  9],
       [16, 25, 36]])
```



Il convient de noter que a et b ont la même forme, deux lignes et trois colonnes. Pour que cette multiplication élément par élément soit possible, il faut en effet que les deux matrices aient la même forme. Dans le cas contraire, un message d'erreur vous avertirait que les formes ne concordent pas. Ici comme dans le cas des vecteurs, la fonction multiply exécute l'opération élément par élément.

Le produit scalaire, en revanche, fonctionne d'une façon totalement différente avec les matrices. Le nombre de colonnes de la matrice a doit être égal au nombre de lignes de la matrice b. Cependant, le nombre de lignes de la matrice a peut être un nombre quelconque et le nombre de colonnes de la matrice b peut être un nombre quelconque, tant qu'il s'agit de multiplier a par b. Ainsi, par exemple, le code suivant donne un produit scalaire correct :

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[1,2,3],[3,4,5],[5,6,7]])

a.dot(b)
array([[22, 28, 34],
       [49, 64, 79]])
```

Il convient de noter que l'output contient le nombre de lignes de la matrice a et le nombre de colonnes de la matrice b. Mais alors, comment tout cela fonctionne-t-il ? Pour obtenir la valeur observée dans le tableau (*array*) résultant à l'indice [0,0], c'est-à-dire 22, on additionne les valeurs de $a[0,0] * b[0,0]$ (c'est-à-dire 1), $a[0,1] * b[1,0]$ (c'est-à-dire 6) et $a[0,2] * b[2,0]$ (c'est-à-dire 15). L'opération est la même pour les autres entrées.

Un avantage de l'utilisation de la classe matrix de NumPy est que certaines tâches deviennent plus évidentes. La multiplication, par



exemple, fonctionne précisément comme on pouvait s'y attendre. Le code suivant génère un produit scalaire en utilisant la classe `matrix` :

```
a = np.mat([[1, 2, 3], [4, 5, 6]])  
b = np.mat([[1, 2, 3], [3, 4, 5], [5, 6, 7]])  
  
a*b  
matrix([[22, 28, 34],  
        [49, 64, 79]])
```

Avec l'opérateur `*`, l'output est le même que lorsque la fonction `dot` est utilisée avec un array. Cet exemple souligne aussi qu'il s'agit de savoir si l'on utilise un objet tableau ou un objet matrice quand on exécute des tâches comme multiplier deux matrices.



Pour effectuer une multiplication élément par élément en utilisant deux objets `matrix`, on doit utiliser la fonction `numpy.multiply`.

Définir des opérations avancées sur les matrices

Ce livre vous propose toutes sortes d'opérations intéressantes sur les matrices, mais vous en utilisez certaines communément, c'est pourquoi elles sont abordées dans ce chapitre. Quand vous travaillez avec des tableaux, vous obtenez parfois des données sous un format qui est incompatible avec l'algorithme. Heureusement, `numpy` s'accompagne d'une fonction spéciale `reshape` qui vous permet de mettre les données sous le format nécessaire, quel qu'il soit. Vous pouvez d'ailleurs vous en servir pour convertir un vecteur en matrice, comme le montre le code suivant :

```
changeIt = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
  
changeIt  
array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
changeIt.reshape(2,4)
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
changeIt.reshape(2,2,2)
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
```



La forme initiale de `changeIt` est celle d'un vecteur, mais la fonction `reshape` en fait une matrice. Par ailleurs, vous pouvez donner à cette matrice le nombre de dimensions approprié compte tenu des données. Il faut seulement que le format de la matrice corresponde au nombre d'éléments requis. Ainsi, par exemple, l'appel de `changeIt.reshape(2,3,2)` sera un échec car il n'y aura pas assez d'éléments pour obtenir une matrice de cette taille.

Dans certaines formulations d'algorithmes, vous pourrez trouver deux opérations importantes sur les matrices. Il s'agit de la transposition et de l'inversion. La *transposition* est la transformation d'une matrice $n \times m$ en matrice $m \times n$, effectuée en échangeant les lignes et les colonnes. Dans la plupart des ouvrages, cette opération est notée A^T . Cette opération est utilisée le plus souvent pour les besoins de la multiplication, en vue d'obtenir les bonnes dimensions. Avec `numpy`, on utilise la fonction `transpose`. Une matrice ayant deux lignes et quatre colonnes, par exemple, sera transposée en matrice de quatre lignes et deux colonnes, comme le montre cet exemple :

```
changeIt
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

np.transpose(changeIt)
array([[1, 5],
       [2, 6],
       [3, 7],
       [4, 8]])
```

L'*inversion matricielle* s'applique aux matrices $m \times m$, ou matrices carrées, c'est-à-dire ayant le même nombre de lignes et

de colonnes. Cette opération est très importante car elle permet la résolution immédiate d'équations faisant intervenir la multiplication de matrices, notamment celles de type $y=bX$ dans lesquelles il s'agit de trouver les valeurs contenues dans le vecteur b . Sachant qu'à la plupart des scalaires (zéro figurant parmi les exceptions) correspond un nombre par lequel la multiplication donne la valeur 1, l'idée est de trouver une matrice inverse par laquelle la multiplication donnera une matrice particulière appelée *matrice identité*. Pour voir apparaître une matrice identité dans numpy, utilisez la fonction `identity` :

```
np.identity(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Il convient de noter que dans une matrice identité, tous les « 1 » sont sur la diagonale. Il est très facile de trouver l'inverse d'un scalaire (le nombre scalaire n a pour inverse n^{-1} , soit $1/n$). Avec les matrices, les choses ne se passent pas de la même manière. L'inversion d'une matrice fait intervenir un grand nombre de calculs. L'inverse d'une matrice A se note A^{-1} . Pour obtenir une matrice inverse avec numpy, utilisez la fonction `linalg.inv`. L'exemple suivant vous montre comment obtenir l'inverse d'une matrice et comment l'utiliser pour obtenir un produit scalaire, puis comparer ce produit scalaire à la matrice identité à l'aide de la fonction `allclose`.

```
a = np.array([[1,2], [3,4]])
b = np.linalg.inv(a)

np.allclose(np.dot(a,b), np.identity(2))
True
```



Trouver l'inverse d'une matrice est parfois impossible. Une matrice qui ne peut pas être inversée est dite *non inversible* ou *singulière*. Les matrices singulières ne sont pas la norme : elles sont très rares.

Créer des combinaisons de la bonne manière

La mise en forme des données implique souvent de les considérer sous différents angles. Les données ne sont pas simplement une séquence de nombres : il existe un ordre logique porteur d'une information pour l'utilisateur. La création des bonnes combinaisons de données grâce à la manipulation des séquences de données est essentielle pour que les algorithmes produisent ce que l'on en attend. Les sections suivantes présentent trois techniques de mise en forme des données : les permutations, les combinaisons et les répétitions.

Distinguer les permutations

Les données brutes arrivent dans un certain ordre, qui peut ne correspondre à rien, sinon à l'ordre d'arrivée des inputs sur une ligne de production. Il peut s'agir, par exemple, d'une série de nombres représentant la quantité de produits fabriqués à un moment donné. La raison pour laquelle les données arrivent dans cet ordre plutôt que dans un autre peut avoir son importance, sans pour autant que cet ordre convienne pour obtenir le résultat que vous attendez d'un algorithme. C'est pourquoi il peut être utile d'effectuer une *permutation* des données, c'est-à-dire de les disposer dans un ordre déterminé par une autre logique.

Les permutations peuvent être envisagées de différentes manières. La présentation aléatoire des données est une des possibilités. En l'occurrence, elle consiste à utiliser la fonction `numpy.random.permutation` :

```
a = np.array([1,2,3])
np.random.permutation(a)
array([2, 3, 1])
```


L'output de votre système pourra être différent de celui qui apparaît ici. À chaque exécution de ce code, les données apparaissent dans un ordre aléatoire, ce qui est pratique lorsqu'un algorithme doit être alimenté par des données randomisées pour donner les résultats désirés. L'échantillonnage est essentiel en analyse de données, notamment, et la technique illustrée ici est un bon moyen de le réaliser.

Dans certains cas, vous pouvez avoir besoin d'obtenir toutes les permutations d'un jeu de données afin de les traiter l'une après l'autre. Pour pouvoir accomplir cette tâche, vous devez importer le module `itertools`. Le code suivant illustre une technique qui vous permet d'obtenir la liste de toutes les permutations d'un vecteur :

```
from itertools import permutations

a = np.array([1,2,3])

for p in permutations(a):
    print(p)

(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```



Pour sauvegarder une telle liste, vous pouvez toujours créer une liste vide et utiliser la fonction `append` pour ajouter chaque série obtenue à cette liste au lieu d'écrire les séries à la suite comme ci-dessus. La liste résultante peut servir d'input à un algorithme conçu pour traiter des jeux de données multiples. Pour plus de détails sur le module `itertools`, consultez la page <https://docs.python.org/3/library/itertools.html>.

Mélanger les combinaisons

Dans certains cas, vous n'avez pas besoin d'un jeu complet mais seulement d'une petite partie des données, sous forme de combinaisons d'une longueur spécifiée. Supposons que vous disposiez d'un jeu de données constitué de quatre nombres et que vous ne vouliez que deux combinaisons de ces nombres (la possibilité d'obtenir certaines parties d'un jeu de données est essentielle pour générer un graphe entièrement connecté, un sujet abordé dans la Troisième partie de ce livre). Le code suivant montre comment obtenir ces combinaisons :

```
from itertools import combinations

a = np.array([1,2,3,4])

for comb in combinations(a, 2):
    print(comb)

(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
```

L'output est constitué de toutes les combinaisons possibles de deux nombres de la matrice `a`. Il convient de noter que cet exemple utilise la fonction `itertools combinations` (la fonction `permutations` est présentée dans la section précédente). Naturellement, vous n'avez pas nécessairement besoin de toutes ces combinaisons : un sous-ensemble aléatoire serait peut-être plus adapté. Dans ce cas, la fonction `random.sample` vous sera utile :

```
pool = []

for comb in combinations(a, 2):
    pool.append(comb)

random.sample(pool, 3)
[(1, 4), (3, 4), (1, 2)]
```

Les combinaisons qui apparaîtront en output ne seront pas toujours les mêmes. Néanmoins, l'idée est que vous avez limité le jeu de données de deux manières : premièrement, en n'utilisant pas toujours tous les éléments de données, et deuxièmement, en n'utilisant pas toutes les combinaisons possibles de ces éléments. Il en résulte une série d'éléments en apparence aléatoire, utilisable comme input d'un algorithme.



Une autre possibilité est de créer une liste complète, mais de rendre l'ordre des éléments aléatoire. La randomisation consiste à mélanger les éléments comme on bat les cartes, et l'on utilise pour cela la fonction `random.shuffle`. Python offre d'ailleurs toute une série de méthodes de randomisation (voir sur la page <https://docs.python.org/3/library/random.html>). Dans un certain nombre d'exemples qui vont suivre dans ce livre, la randomisation est aussi utilisée pour obtenir un output correct des algorithmes.

Résoudre le problème des répétitions

Des données répétées peuvent affecter négativement l'output d'un algorithme et rendre les résultats inexploitable. Il est parfois nécessaire de disposer de valeurs non répétées avant de procéder à toute manipulation des données. Heureusement, avec Python, il est facile d'éliminer certains types de redondances, comme dans cet exemple :

```
a = np.array([1,2,3,4,5,6,6,7,7,1,2,3])
b = np.array(list(set(a)))

b
array([1, 2, 3, 4, 5, 6, 7])
```



Ici, `a` présente initialement une série de nombres qui ne sont dans aucun ordre particulier et dans laquelle on constate des répétitions. Or, dans Python, un jeu de données ne comporte jamais de données répétées. En convertissant cette liste en `set` (ensemble) puis à nouveau en `list` (liste), puis en plaçant cette liste dans un `array` (tableau), on obtient un vecteur dans lequel il n'y a plus de

répétitions.

Obtenir les résultats désirés grâce à la récursivité

La récursivité est une méthode de programmation sophistiquée pour résoudre un certain nombre de problèmes en informatique. Elle repose sur la capacité d'une fonction à s'appeler elle-même de façon répétitive jusqu'à ce qu'une certaine condition soit satisfaite. Le terme *récursivité* vient du verbe latin *recurrere* qui signifie revenir.

La récursivité consiste à appeler une même fonction un certain nombre de fois, mais en modifiant les paramètres de l'appel. Cette méthode est utilisée surtout parce qu'elle facilite la résolution des problèmes en algorithmique en imitant la manière dont l'être humain les résoudrait. Malheureusement, ce n'est pas un outil d'utilisation facile. Il faut arriver à comprendre comment il est possible de créer un sous-programme récursif en prévoyant des paramètres qui éviteront les problèmes de mémoire insuffisante sur l'ordinateur. Les sections qui suivent expliquent en détail le fonctionnement de la récursivité et présentent un exemple d'application avec Python.

Expliquer la récursivité

L'utilisation de la récursivité pose un problème à beaucoup de gens, qui ont des difficultés à s'en représenter le fonctionnement. Dans la plupart de cas, quand vous appelez une fonction dans un programme, cette fonction exécute une tâche donnée puis s'arrête. La récursivité consiste à appeler une fonction qui va exécuter une tâche, puis s'appeler elle-même de façon répétitive jusqu'à la réalisation d'une condition particulière, les appels qui précèdent restant actifs. Ces appels se débouclent l'un après l'autre jusqu'à ce que le premier appel se termine finalement avec la réponse

correcte, et c'est ce processus de débouclage qui présente des difficultés pour beaucoup d'utilisateurs. La [Figure 5-1](#) montre à quoi ressemble la récursivité quand on utilise un diagramme de flux.

Il importe de noter la condition au centre du processus. Elle est indispensable à la viabilité de la récursivité. Sans cette condition, on obtiendrait une boucle sans fin. La condition est un test pouvant avoir deux résultats possibles :

- » soit les conditions pour mettre fin à la récursivité ne sont pas encore remplies, et la fonction doit s'appeler elle-même à nouveau ;

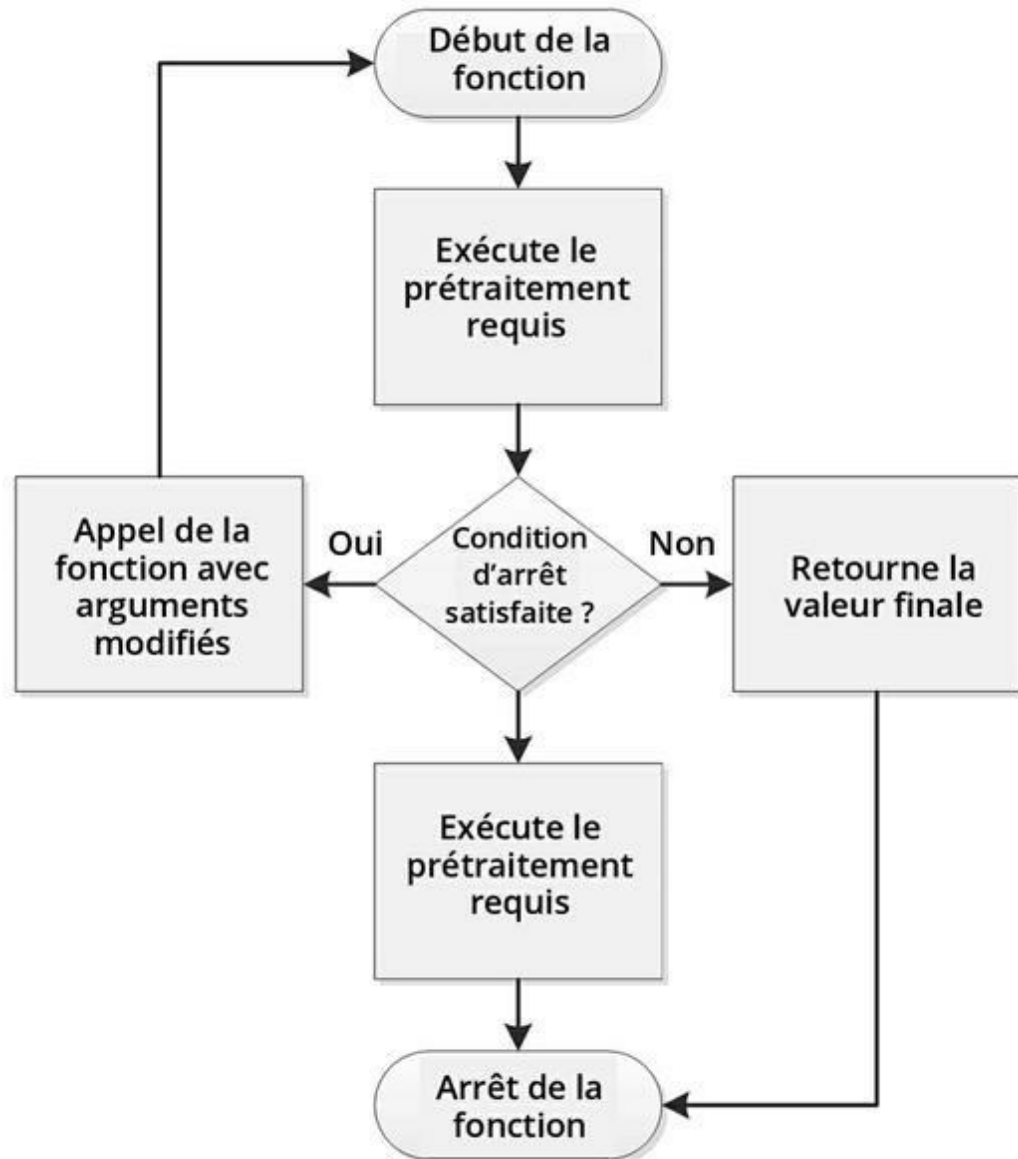


FIGURE 5-1 Dans un processus de récursivité, une fonction s'appelle elle-même continuellement jusqu'à ce qu'une condition donnée soit satisfaite.

- » soit les conditions pour mettre fin à la récursivité sont remplies, et la fonction retourne une valeur finale qui va servir à calculer le résultat final.



Quand une fonction s'appelle elle-même, elle n'utilise pas continuellement les mêmes arguments qui lui ont été transmis : ou alors, la condition ne changerait jamais et la récursivité n'aurait jamais de fin. La récursivité suppose donc que les appels successifs de la fonction modifient les arguments de manière à conduire la fonction vers une solution.

Un des exemples de récursivité les plus courants, tous langages de programmation confondus, est le calcul d'une factorielle. Une *factorielle* est la multiplication d'une série de nombres entre une valeur de départ et une valeur d'arrivée, sachant que chaque nombre de la série est égal au nombre qui le précède moins un. Ainsi, par exemple, pour calculer 5 ! (lire « factorielle cinq »), on effectue la multiplication $5 * 4 * 3 * 2 * 1$. Ce calcul est un parfait exemple de récursivité, aussi simple soit-il. Le code Python permettant ce calcul est le suivant (vous le retrouverez dans le fichier téléchargeable A4D ; 05 ; Recursion.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
def factorial(n):
    print("factorial appelée pour n = ", str(n))
    if n == 1 or n == 0:
        print("Condition d'arrêt satisfaite.")
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))

factorial appelée pour n = 5
factorial appelée pour n = 4
factorial appelée pour n = 3
factorial appelée pour n = 2
factorial appelée pour n = 1
Condition d'arrêt satisfaite.
120
```

Le code satisfait la condition d'arrêt quand $n == 1$. Chacun des appels successifs de la fonction factorial utilise factorial(n-1), si

bien que l'argument de départ diminue à chaque fois de 1. L'output affiche chaque appel puis indique que la condition d'arrêt est satisfaite. Le résultat, 120, est égal à 5 ! (factorielle cinq).

Il est important de comprendre qu'il n'existe pas une méthode unique de résolution de problème par l'utilisation de la récursivité. Comme pour n'importe quelle autre technique de programmation, on peut trouver toutes sortes de moyens d'accomplir la même chose. Voici, par exemple, une autre version du calcul factoriel par la récursivité, qui utilise moins de lignes de code tout en exécutant aussi efficacement la même tâche :

```
def factorial(n):
    print("factorial appelée pour n = ", str(n))
    if n > 1:
        return n * factorial(n-1)
    print("Condition d'arrêt satisfaite.")
    return 1
```

```
print(factorial(5))
```

```
factorial appelée pour n = 5
factorial appelée pour n = 4
factorial appelée pour n = 3
factorial appelée pour n = 2
factorial appelée pour n = 1
Condition d'arrêt satisfaite.
120
```



Notez bien la différence. Au lieu de tester la condition d'arrêt, cette version teste la condition de continuation. Tant que n est supérieur à 1, le programme continue à exécuter des appels récursifs. Bien qu'il soit plus court que dans la version précédente, ce code est moins clair, car à présent il faut réfléchir pour savoir quelle condition mettra fin à la récursivité.

Éliminer la récursivité terminale

De nombreuses formes de récursivité utilisent la récursivité terminale. C'est même le cas de l'exemple de la section précédente. La *récursivité terminale* consiste à appeler une fonction en dernière instruction avant le retour du résultat. Dans la section précédente, la ligne `return n * factorial(n-1)` est la « récursion » terminale.

La récursivité terminale n'est pas nécessairement une mauvaise chose. C'est de cette façon que les programmeurs écrivent les sous-programmes récursifs la plupart du temps. Cependant, le recours à la récursivité terminale oblige Python à conserver en mémoire chaque valeur d'appel jusqu'au débouclage de la récursivité. Or, chaque appel consomme de la mémoire. À un moment donné, la mémoire est saturée et l'appel est un échec, si bien que l'algorithme ne fonctionne plus. Étant donné la complexité de certains algorithmes actuels et la taille considérable des jeux de données qu'ils utilisent, la récursivité terminale peut occasionner de sérieux déboires aux utilisateurs.

Avec un peu d'imagination, vous pouvez éliminer la récursivité terminale de vos sous-programmes récursifs. Vous trouverez en ligne tout un tas de techniques vraiment remarquables, comme l'utilisation d'un trampoline, comme l'explique la page <http://blog.moertel.com/posts/2013-06-12-recursion-to-iteration-4-trampolines.html>. Néanmoins, la méthode la plus simple, pour éliminer cette récursivité, consiste à créer une alternative itérative pour exécuter la même tâche. Voici, par exemple, une fonction `factorial` qui utilise le principe des itérations de préférence à la récursivité pour éviter les éventuels problèmes de saturation de la mémoire :

```
def factorial(n):
    print("factorial appelée pour n = ", str(n))
    result = 1
    while n > 1:
        result = result * n
        n = n - 1
        print("La valeur courante de n est ", str(n))
    print("Condition d'arrêt satisfaite.")
    return result
```

```
print(factorial(5))
```

```
factorial appelée pour n = 5  
La valeur courante de n est 4  
La valeur courante de n est 3  
La valeur courante de n est 2  
La valeur courante de n est 1  
Condition d'arrêt satisfaite  
120
```

Le déroulement de cette fonction est le même que celui de la fonction récursive. Une boucle `while` remplace l'appel récursif, mais il faut encore vérifier la même condition et continuer la boucle jusqu'à ce que les données satisfassent la condition. Le résultat est le même. Cependant, dans certains cas, il n'est pas évident de remplacer la récursivité par l'itération, comme on peut le voir dans l'exemple de la page <http://blog.moertel.com/posts/2013-06-03-recursion-toiteration-3.html>.

Exécuter les tâches plus rapidement

Bien évidemment, l'idéal est toujours que les tâches soient exécutées le plus rapidement possible. Cependant, il importe de comparer attentivement les alternatives. Renoncer à une partie de la mémoire pour qu'une tâche s'exécute plus vite est une bonne chose tant qu'il reste suffisamment de mémoire. Les chapitres suivants recensent les différentes manières d'accélérer l'exécution des tâches, mais vous pouvez essayer les principales techniques à tout moment, quel que soit l'algorithme sur lequel vous travaillez. Les sections suivantes étudient certaines de ces techniques.

Diviser pour régner

Certains problèmes paraissent insurmontables au début. Prenons l'exemple de la rédaction d'un livre. Si vous songez à l'intégralité de l'ouvrage, la tâche vous semblera considérable. En revanche, si

vous divisez le livre en chapitres et si vous ne vous intéressez pour le moment qu'à un seul de ces chapitres, le travail vous paraîtra déjà plus faisable. Bien sûr, écrire un chapitre entier peut encore vous sembler quelque peu rebutant : dans ce cas, continuez à décomposer la tâche. Divisez le chapitre en titres de premier niveau. Cela ne suffira peut-être pas. Divisez chaque partie de chapitre en notant des titres de second niveau, et ainsi de suite, jusqu'à ce que le chapitre soit constitué d'articles aussi courts que possible. Si même un court article reste une tâche difficile, divisez-le en paragraphes, puis en phrases, et finalement, en mots. Écrire un seul mot n'est pas difficile. Ainsi, écrire un livre revient à écrire des mots : beaucoup de mots. C'est de cette manière que l'on applique le principe de diviser pour régner, en décomposant un problème en problèmes plus petits jusqu'à obtenir un problème que vous puissiez résoudre sans trop de difficulté.

Les ordinateurs aussi peuvent recourir à cette méthode. La résolution d'un problème gigantesque associé à un énorme jeu de données peut prendre plusieurs jours, en supposant que la tâche soit réalisable. Cependant, en décomposant ce problème en parties plus petites, il est possible de le résoudre beaucoup plus vite et en consommant moins de ressources. Si vous cherchez une donnée dans une base de données, par exemple, il n'est pas nécessaire de parcourir toute la base, du moins, si son contenu est ordonné. Supposons que vous y recherchiez le mot *Hello*. Vous pourriez commencer par diviser la base en deux (lettres *A* à *M* et lettres *N* à *Z*). La valeur numérique du *H* de *Hello* (72 si l'on se réfère à une table ASCII standard) étant inférieure à celle du *M* (77) dans l'alphabet, vous pouvez limiter votre recherche à la première moitié de la base de données. Divisez encore cette moitié (lettres *A* à *G* et *H* à *M*), et c'est maintenant la seconde moitié de ce reste qui vous intéresse : elle ne représente que le quart de la base de données. Vous pouvez poursuivre ce découpage afin de limiter votre recherche à une petite partie seulement de l'ensemble de la base. Cette méthode est ce que l'on appelle la *recherche binaire*. Il s'agit donc de procéder comme suit :

- 1. Divisez en deux le contenu en question.**

2. **Comparez les clés pour le contenu avec le terme de recherche.**
3. **Entre les deux moitiés obtenues, choisissez celle qui contient la clé.**
4. **Répétez les étapes 1 à 3 jusqu'à ce que vous trouviez la clé.**



Lorsqu'il s'agit de procéder par division pour réduire les difficultés, l'approche est similaire pour la plupart des problèmes, même si la méthode devient parfois vraiment compliquée. Ainsi, par exemple, au lieu de simplement diviser la base de données en deux, vous pouvez dans certains cas la diviser en trois, mais l'objectif sera toujours le même : diviser le problème en éléments plus petits et déterminer si l'on peut le résoudre en s'intéressant à un seul élément et en généralisant. Par la suite, à partir de cet élément, on peut résoudre le reste aussi. Le code suivant est une version extrêmement simple d'une recherche binaire sur une liste supposée triée (vous pouvez retrouver ce code dans le fichier téléchargeable A4D ; 05 ; Binary Search.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
def search(searchList, key):
    mid = int(len(searchList) / 2)
    print("Recherche du point médian à ",
          str(searchList[mid]))

    if mid == 0:
        print("Clé non trouvée !")
        return key

    elif key == searchList[mid]:
        print("Clé trouvée !")
        return searchList[mid]

    elif key > searchList[mid]:
        print("searchList contient maintenant ",
              searchList[mid:len(searchList)])
        search(searchList[mid:len(searchList)], key)

    else:
        print("searchList contient maintenant ",
```

```
        searchList[0:mid])
    search(searchList[0:mid], key)

aList = list(range(1, 21))
search(aList, 5)

Recherche du point médian à 11
searchList contient maintenant [1, 2, 3, 4, 5, 6, 7,
8, 9, 10]
Recherche du point médian à 6
searchList contient maintenant [1, 2, 3, 4, 5]
Recherche du point médian à 3
searchList contient maintenant [3, 4, 5]
Recherche du point médian à 4
searchList contient maintenant [4, 5]
Recherche du point médian à 5
Clé trouvée !
```

L'approche récursive de la recherche binaire commence avec aList qui contient les nombres 1 à 20. Le programme recherche la valeur 5 dans aList. Chaque itération commence par une recherche du point médian de la liste, mid, et ce point médian est utilisé pour déterminer l'étape suivante. Quand la clé key correspond au point médian, la valeur est trouvée dans la liste et la recherche récursive est terminée.



Dans cet exemple, il convient de noter qu'une seule des deux récursions terminales est utilisée. Quand key est supérieure au point médian de la liste existante searchList[mid], le code appelle à nouveau search avec seulement le terme de droite de ce qui reste de la liste. En d'autres termes, chaque appel à search utilise seulement la moitié de la liste obtenue dans l'appel précédent. Lorsque key est inférieure ou égale à searchList[mid], search reçoit la moitié gauche de la liste existante.



La liste ne contient pas nécessairement la valeur cherchée. Par conséquent, il faut toujours prévoir une méthode de sortie lorsqu'il y a récursivité, faute de quoi la pile se remplira et il s'ensuivra un message d'erreur. En l'occurrence, la sortie se produit lorsque mid == 0, ce qui signifie qu'il n'y a plus de searchList à rechercher. Si vous remplacez search(aList, 5) par search(aList, 22), par exemple, vous obtiendrez cette fois le

résultat suivant :

```
Recherche du point médian à 11
searchList contient maintenant [11, 12, 13, 14, 15,
16, 17, 18, 19,
20]
Recherche du point médian à 16
searchList contient maintenant [16, 17, 18, 19, 20]
Recherche du point médian à 18
searchList contient maintenant [18, 19, 20]
Recherche du point médian à 19
searchList contient maintenant [19, 20]
Recherche du point médian à 20
searchList contient maintenant [20]
Recherche du point médian à 20
Clé non trouvée !
```

Il convient de noter également que le programme recherche la condition de sortie avant d'exécuter toute autre tâche, afin que le code n'entraîne pas une erreur par inadvertance par absence de contenu de searchList. Quand on utilise la récursivité, il faut rester proactif ou s'attendre à en supporter les conséquences.

Distinguer les différentes solutions possibles

La récursivité fait partie des diverses solutions de programmation algorithmique, comme on le verra dans les prochains chapitres. Il est même difficile d'y échapper, sachant que dans de nombreux cas, l'approche itérative se révèle contre-intuitive, peu pratique et coûteuse en temps. Cependant, vous pouvez créer un certain nombre de versions différentes de la même solution, chacune avec ses propres caractéristiques, ses points faibles et ses points forts.

La solution que ce chapitre n'étudie pas est la recherche séquentielle, car une recherche séquentielle prend généralement davantage de temps que toutes les autres solutions envisageables. Dans le meilleur des cas, une seule comparaison suffit pour terminer la recherche, mais dans le pire des cas, il faut attendre le

dernier test pour trouver l'élément voulu. En moyenne, une recherche séquentielle nécessite $(n+1)/2$ tests ou la quantité de temps $O(n)$.

Dans la section précédente, la recherche binaire est bien plus efficace que la recherche séquentielle. Elle utilise le temps logarithmique, $O(\log n)$. Dans le meilleur des cas, un seul test suffit, comme dans une recherche séquentielle, mais le résultat de notre exemple montre que même dans le cas le plus défavorable, lorsque la valeur n'apparaît même pas dans la liste, il ne faut pas plus de six tests, au lieu des 21 tests que nécessiterait une recherche séquentielle.



Ce livre couvre une grande variété d'algorithmes de recherche et de tri, sachant que la recherche et le tri sont deux catégories majeures de traitement informatique. Songez au temps que vous consacrez chaque jour à rechercher des données sur Google. En théorie, vous devriez passer des journées entières à ne faire que cela. Les sous-programmes de recherche fonctionnant mieux avec des données triées, on comprend la nécessité des recherches efficaces et des routines de tri. Heureusement, il n'est pas nécessaire de passer des heures à chercher quels programmes de recherche et de tri seront les plus efficaces. Sur des sites comme Big-O Cheat Sheet, <http://bigocheatsheet.com/>, vous trouverez les informations nécessaires pour déterminer quelle solution est la plus adaptée.



Si vous ne vous préoccupez que du temps d'exécution, les données que vous recevrez risquent de vous induire en erreur et de vous faire croire, à tort, qu'une certaine solution conviendra particulièrement bien pour votre application. Il importe de tenir compte également du type de données avec lequel vous travaillez, de la complexité de l'élaboration de la solution, et de bien d'autres facteurs encore. C'est pourquoi les exemples qui vont suivre dans ce livre tiennent compte des avantages et des inconvénients de chaque méthode, et des dangers cachés d'une solution qui semblerait prometteuse, mais qui ne produirait finalement pas le résultat désiré.

PARTIE 2

L'importance du tri et de la recherche de données

DANS CETTE PARTIE...

Utiliser diverses structures de données dans Python

Travailler sur des arborescences et des graphes

Trier les données pour accélérer l'exécution des algorithmes

Rechercher des données pour localiser avec précision et rapidement la bonne information

Recourir à des techniques de hachage pour créer des index de données plus petits

Chapitre 6

Structurer les données

DANS CE CHAPITRE

- » Comprendre pourquoi structurer les données est nécessaire
 - » Travailler avec des piles, des files, des listes et des dictionnaires
 - » Utiliser des arborescences pour organiser les données
 - » Représenter par des graphes les données et les relations qui les lient
-

Les données brutes, comme leur nom l'indique, ne sont pas structurées ni affinées de quelque manière que ce soit. Il se peut que certaines données soient manquantes, ou endommagées, ou simplement, qu'elles ne conviennent pas. À partir du moment où les données sont brutes, on ne peut pas être sûr de ce que l'on obtiendra et de ce que l'on pourra en faire.

La plupart du temps, avant toute chose, vous devez structurer les données d'une manière ou d'une autre avant de commencer à voir ce qu'elles contiennent (et parfois, ce qu'elles ne contiennent pas). Structurer les données signifie les organiser d'une certaine manière de telle sorte qu'elles aient les mêmes caractéristiques, la même apparence et les mêmes composantes. Ainsi, par exemple, il se peut que vous obteniez des données provenant d'une source comportant des dates sous forme de chaînes et d'une autre source qui utilise des dates sous forme d'objets. Pour pouvoir exploiter ces informations, vous devrez alors faire correspondre les types de données. Les sources de données peuvent aussi structurer les données de façon différente. Il se peut qu'une source stocke le

nom de famille et le prénom dans un champ unique, tandis qu'une autre source conditionne la même information sous forme de deux champs distincts. L'organisation constitue une partie importante de la structuration des données. Il ne s'agit pas du tout de modifier les données, seulement de les rendre plus exploitables (structurer les données n'est pas la même chose que les corriger ou les mettre en forme en changeant éventuellement certaines valeurs pour convertir un type de données en un autre, avec parfois une perte de précision, par exemple avec les dates, quand on doit passer d'une source de données à une autre).

Python donne accès à un certain nombre de structures organisationnelles de données. Ces structures, plus particulièrement les piles, les files et les dictionnaires, sont utilisées dans un grand nombre d'exemples dans ce livre. Chaque structure de données représente un moyen différent de travailler avec les données et un jeu de données différent pour exécuter des tâches comme trier les données dans un certain ordre. Ce chapitre présente les méthodes organisationnelles les plus courantes, notamment les arborescences et les graphes (deux méthodes si importantes qu'une section entière est consacrée à chacune d'elles).

Pourquoi les données doivent être structurées

La structure est un élément essentiel au fonctionnement des algorithmes. Comme le montre l'exemple de la recherche binaire du [Chapitre 5](#), il est beaucoup plus facile de développer un algorithme en utilisant des données structurées que de chercher comment interpréter les données dans le code. Dans la recherche binaire, par exemple, les données doivent se présenter de façon ordonnée. Se lancer dans les comparaisons nécessaires en travaillant sur des données qui se présentent dans le désordre demanderait beaucoup plus d'efforts et la tâche pourrait même se révéler impossible. C'est pour ces raisons qu'il est nécessaire

d'étudier les exigences structurelles relatives aux données utilisées avec les algorithmes, un sujet auquel les sections qui suivent sont consacrées.

Faciliter la visualisation du contenu

Dans l'exploitation des données, il est essentiel de comprendre ce qu'elles contiennent. Pour qu'un algorithme de recherche puisse produire un résultat, il faut d'abord que vous ayez compris en quoi consiste le jeu de données et que vous sachiez quoi rechercher. Rechercher des mots dans un jeu de données qui contient des nombres est une tâche impossible qui donne toujours lieu à des erreurs. Pourtant, les erreurs de recherche dues à une mauvaise perception du contenu d'un jeu de données sont chose courante, même avec les meilleurs moteurs de recherche. Les suppositions des utilisateurs concernant le contenu des données entraînent un dysfonctionnement des algorithmes. Par conséquent, mieux vous percevrez le contenu des données grâce à une présentation structurée, plus il vous sera facile de mener à bien des tâches au moyen d'algorithmes.

Néanmoins, même l'examen du contenu est souvent source d'erreurs en raison de la différence de perception entre utilisateur et machine. Si vous recherchez un nombre formaté sous forme d'une chaîne alors que le jeu de données est constitué de nombres formatés sous forme d'entiers, la recherche n'aboutira pas. Un ordinateur ne traduit pas automatiquement une chaîne en nombre entier, ou l'inverse, comme nous le faisons nous autres humains. L'ordinateur ne perçoit que des nombres, et les chaînes ne sont qu'une interprétation imposée aux nombres par le programmeur. Si vous recherchez « 1 » (une chaîne), l'ordinateur interprétera votre demande comme une recherche du nombre 49, dans le contexte de l'utilisation des caractères ASCII. Pour trouver la valeur numérique 1, vous devez rechercher 1 en tant que valeur numérique (nombre entier).



La structuration des données est aussi ce qui vous permet d'étudier la subtilité du format des données. Un numéro de téléphone, par exemple, peut apparaître sous la forme (41)-223275148. Si vous effectuez une recherche ou une autre tâche algorithmique sous la forme 00(41)- 223275148, la recherche risque d'être infructueuse en raison de l'addition de 00 au début du terme de recherche. Ce genre de détail peut être source de problèmes importants, car si ces deux formes peuvent sembler équivalentes aux yeux de la plupart des gens, elles ne le sont pas pour l'ordinateur. La machine les considère comme deux formes bien différentes, et même, de longueur différente. Imposer une forme donnée aux humains est souvent vain et entraîne généralement une frustration qui rend l'utilisation de l'algorithme plus difficile encore, aussi une structure imposée par le biais d'une manipulation des données devient plus importante encore.

Homogénéiser des données provenant de sources différentes

Interagir avec des données provenant d'une source unique est un problème, mais interagir avec des données provenant de plusieurs sources en est un autre. Or, de nos jours, les jeux de données proviennent généralement de plus d'une source, et il importe de se rendre compte des complications que cela peut engendrer. Quand vous utilisez deux jeux de données qui ne proviennent pas de la même source, procédez comme suit :

- » Déterminez si les deux jeux de données contiennent toutes les données nécessaires. Il y a peu de chances pour que deux fournisseurs différents créent des jeux de données contenant précisément les mêmes données, de même type, sous le même format et dans le même ordre. Par conséquent, il importe de savoir si vous disposez des données dont vous avez besoin ou si vous devez effectuer des changements pour obtenir le résultat désiré, comme nous allons le voir dans la section suivante.

- » Vérifiez le type des données transmises par chacune des deux sources. Il se peut, par exemple, que les dates soient sous forme de chaînes dans un jeu de données, et sous forme de dates dans l'autre. Les incohérences entre les types de données seront source de problèmes si un algorithme qui traite les dates sous une certaine forme les reçoit sous une autre forme.
- » Assurez-vous que les éléments de données ont la même signification dans les deux jeux de données. La taille d'un entier, par exemple, peut varier d'une source à l'autre. Les entiers peuvent être codés sur 16 bits d'un côté, et sur 32 bits de l'autre. Les valeurs faibles auront la même signification, mais le codage sur 32 bits permet de gérer des valeurs plus grandes, ce qui peut entraîner un problème pour votre algorithme. Les dates aussi peuvent poser des problèmes, sachant qu'elles stockent souvent un nombre considérable de millisecondes depuis une date donnée (JavaScript, par exemple, comptabilise le nombre de millisecondes écoulées depuis le 1^{er} janvier 1970 UTC). Pour l'ordinateur, il n'existe que des nombres : ce sont les utilisateurs qui donnent du sens à ces nombres, de telle sorte que les applications les interprètent de manière spécifique.
- » Vérifiez les attributs des données. Les éléments de données ont des attributs spécifiques, et c'est la raison pour laquelle le [Chapitre 4](#) vous explique comment Python interprète les différents types de données. Le [Chapitre 5](#) montre comment cette interprétation peut changer quand on utilise numpy. Les attributs de données changent d'un environnement à un autre, et les développeurs peuvent les changer davantage encore en créant des types de données personnalisés. Pour pouvoir combiner des données provenant de sources différentes, vous devez prendre en compte ces attributs, de telle sorte qu'elles soient interprétées correctement.



Plus vous consacrerez du temps à vérifier la compatibilité des données des différentes sources à utiliser pour votre jeu de données, moins vous risquerez de rencontrer des problèmes en travaillant avec un algorithme. Les problèmes d'incompatibilité entre les données n'apparaissent pas toujours comme des erreurs. Dans certains cas, une incompatibilité peut poser d'autres problèmes, notamment, donner des résultats qui semblent corrects mais qui sont erronés.

Par ailleurs, combiner des données provenant de plusieurs sources ne signifie pas nécessairement créer un nouveau jeu de données apparemment semblables à celles provenant de ces sources. Dans certains cas, vous créez des agrégats de données ou vous effectuez d'autres formes de manipulations pour créer de nouvelles données à partir des données existantes. L'analyse peut prendre toutes sortes de formes, et les formes les plus inhabituelles peuvent entraîner des erreurs très regrettables si elles sont utilisées de façon incorrecte. Supposons, par exemple, qu'une source de données fournisse des informations générales sur les clients de l'entreprise et qu'une seconde source de données fournisse leurs habitudes d'achat. Par suite d'un décalage entre les deux sources, les clients pourraient se voir attribuer des habitudes d'achat qui ne sont pas les leurs, ce qui engendrerait des problèmes lorsque vous leur proposeriez de nouveaux produits. Prenons un exemple extrême : imaginons ce qui pourrait se produire si les informations sur des patients provenant de plusieurs sources étaient rassemblées sous forme d'une nouvelle source de données, mais avec toutes sortes de décalages. Un patient sans antécédents particuliers pourrait se voir attribuer le diagnostic d'une maladie qu'il n'a jamais eue et un historique de soins qu'il n'a jamais reçus.

Quand il devient nécessaire de procéder à des changements

Si vos jeux de données posent des problèmes, il est nécessaire d'y remédier de telle sorte que ces jeux de données puissent être traités correctement par vos algorithmes. En cas d'hétérogénéité des types de données, par exemple, vous devrez changer les types de données dans chaque source de données afin de les homogénéiser, puis créer la source de données unique que vous utiliserez avec votre algorithme. La tâche prend du temps mais pour l'essentiel, elle n'est pas compliquée. Avant de procéder, vous devez simplement avoir une bonne compréhension des données, c'est-à-dire une bonne perception de leur contenu dans le contexte de ce que vous comptez en faire. Néanmoins, il importe d'envisager deux situations particulières : les données dupliquées, et les données manquantes. Les sections qui suivent montrent comment résoudre ces problèmes.

Quand des données sont dupliquées

Des données peuvent se trouver dupliquées pour un certain nombre de raisons. Certaines sont évidentes : un utilisateur peut avoir, par inadvertance, saisi la même donnée plus d'une fois. Par distraction, il arrive qu'un utilisateur ne sache plus où il en était dans la saisie d'une liste. Il peut arriver également que deux utilisateurs saisissent le même enregistrement. L'origine d'une duplication est parfois moins évidente. Combiner deux ou plusieurs jeux de données peut entraîner des duplications, lorsque certains enregistrements leur sont communs. Des duplications de données peuvent aussi résulter de l'utilisation de diverses techniques de mise en forme des données, lorsqu'il s'agit de créer de nouvelles données à partir des sources de données existantes. Heureusement, des modules comme Pandas vous permettent de supprimer les données dupliquées, comme le montre l'exemple suivant (vous pouvez retrouver ce code dans le fichier téléchargeable A4D ; 06 ; Remediation.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import pandas as pd

df = pd.DataFrame({'A': [0,0,0,0,0,1,0],
```

```

        'B': [0, 2, 3, 5, 0, 2, 0],
        'C': [0, 3, 4, 1, 0, 2, 0]})
print(df, "\n")

```

```

df = df.drop_duplicates()
print(df)

```

```

A B C
0 0 0 0
1 0 2 3
2 0 3 4
3 0 5 1
4 0 0 0
5 1 2 2
6 0 0 0
A B C
0 0 0 0
1 0 2 3
2 0 3 4
3 0 5 1
5 1 2 2

```

Dans cet exemple, la fonction `drop_duplicates` supprime les doublons trouvés sur les lignes 4 et 6. Une lecture des données dans le pandas DataFrame permet la suppression rapide des entrées qui font doublon et qui auraient grevé inutilement l'output de l'algorithme de traitement.

Quand certaines données sont manquantes

Le fait que des données soient manquantes peut aussi fausser les résultats de l'exécution d'un algorithme. Certains algorithmes peuvent produire un résultat inattendu, ou même générer une erreur. En résumé, sachant que des valeurs manquantes peuvent engendrer des problèmes de données, il importe d'y remédier. Vous avez pour cela plusieurs possibilités. Vous pouvez attribuer aux données non renseignées une valeur standard, par exemple 0 pour les entiers. Bien sûr, cela peut aussi fausser les résultats. Une autre méthode consiste à remplacer les valeurs manquantes par la moyenne de toutes les valeurs dont on dispose. C'est un peu comme si les valeurs manquantes ne comptaient pas. C'est la méthode qui a été utilisée dans l'exemple suivant :


```

import pandas as pd
import numpy as np

df = pd.DataFrame({'A': [0,0,1,None],
                    'B': [1,2,3,4],
                    'C': [np.NaN,3,4,1]},
                  dtype=int)
print(df, "\n")

values = pd.Series(df.mean(), dtype=int)
print(values, "\n")

df = df.fillna(values)
print(df)

A B C
0 0 1 NaN
1 0 2 3
2 1 3 4
3 None 4 1

A 0
B 2
C 2
dtype: int32

A B C
0 0 1 2
1 0 2 3
2 1 3 4
3 0 4 1

```

La fonction `fillna` vous permet d'éliminer les valeurs qui font défaut, qu'il s'agisse de valeurs non numériques (NaN) ou simplement manquantes (None). Il y a plusieurs manières de combler les trous. Dans cet exemple, nous avons une série contenant la moyenne sur chaque colonne (la méthode est similaire à celle qui serait employée en travaillant avec une base de données).



Il convient de noter que le code se garde bien d'introduire des erreurs dans l'output, en faisant en sorte que `values` soit du type de données approprié. Normalement, la fonction `mean` donne des valeurs à virgule flottante, il est possible d'imposer le type de

données adéquat à cette série. En conséquence, non seulement l'output ne comporte plus de valeurs manquantes, mais il comporte des données du type voulu.

Envisager d'autres problèmes de correction des entrées

La correction des données en entrée peut prendre d'autres formes. Il peut arriver qu'un utilisateur prépare un input incohérent ou incorrect. Toutes les applications n'imposent pas à l'utilisateur des règles strictes pour la saisie de données, si bien qu'il est parfois possible de saisir des noms de pays incorrects, par exemple. Il y a également les erreurs d'orthographe. Il peut arriver aussi que des valeurs soient hors limite, ou impossibles dans une situation donnée. Il n'est pas toujours possible d'épurer complètement les données à la première tentative. Souvent, on ne peut se rendre compte d'un problème qu'en exécutant l'algorithme et en remarquant une anomalie au niveau des résultats, ou en constatant que l'algorithme ne fonctionne pas (même s'il a pu traiter une partie des données). En cas de doute, vérifiez vos données.

Empiler les données dans le bon ordre

Python offre un certain nombre de méthodologies de stockage, qui sont évoquées au [Chapitre 4](#). Comme on l'a vu au [Chapitre 5](#) ainsi que dans ce chapitre, les modules complémentaires proposent souvent d'autres méthodes encore. NumPy et Pandas offrent des alternatives en matière de stockage, et vous pourriez y recourir utilement lorsque vous êtes confronté à divers problèmes de structuration des données.



Un problème courant en matière de stockage des données est la nécessité, outre le besoin de stockage de données, de stocker ces données dans un ordre particulier de manière à pouvoir y accéder

à volonté. Ainsi, par exemple, vous pouvez avoir besoin d'être sûr que le premier élément placé sur une pile d'éléments à traiter sera effectivement le premier élément traité. Les sections qui suivent présentent les méthodes de Python permettant un stockage des données dans le bon ordre pour les besoins spécifiques du traitement.

Classer les données en piles

La pile est un stockage de données de type LIFO (*last in/first out*, c'est-à-dire dernier entré, premier sorti). Le module NumPy permet une véritable implémentation de piles. Par ailleurs, Pandas associe des piles à des objets comme le DataFrame. Cependant, ces deux modules ne font pas apparaître les détails de la mise en œuvre des piles, or il est vraiment utile de comprendre comment une pile fonctionne. C'est pourquoi l'exemple qui suit met en œuvre une pile en utilisant une liste (list) standard de Python (vous retrouverez ce code dans le fichier téléchargeable A4D ; 06 ; Stacks, Queues, and Dictionaries.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
MyStack = []
StackSize = 3

def DisplayStack():
    print("La pile contient maintenant :")
    for Item in MyStack:
        print(Item)

def Push(Value):
    if len(MyStack) < StackSize:
        MyStack.append(Value)
    else:
        print("Pile pleine !")

def Pop():
    if len(MyStack) > 0:
        print("Dépilage : ", MyStack.pop())
    else:
        print("La pile est vide.")
Push(1)
```

```
Push(2)
Push(3)
DisplayStack()
```

```
Push(4)
```

```
Pop()
DisplayStack()
```

```
Pop()
Pop()
Pop()
```

La pile contient maintenant :

1

2

3

Pile pleine !

Dépilage : 3

La pile contient maintenant :

1

2

Dépilage : 2

Dépilage : 1

La pile est vide.

Dans cet exemple, la pile maintient l'intégrité des données et celles-ci sont traitées dans l'ordre prévu. Le code effectue une simple manipulation de liste (list), mais cette méthode est efficace et une telle représentation sous forme de pile peut être utilisée en toutes circonstances.



Avec Python, les listes sont des listes ordonnées de valeurs et leur utilisation est facile et intuitive. Du point de vue algorithmique, elles sont souvent peu adaptées, car les éléments sont stockés dans la mémoire de l'ordinateur et leur accès se fait à l'aide d'un index et de pointeurs (un pointeur étant un nombre qui représente l'adresse de la donnée en mémoire). La logique de fonctionnement est exactement la même que celle de l'index d'un livre. Les listes n'ont pas connaissance de leur contenu. Quand votre application lance une demande de données, le programme parcourt tous les éléments de la liste, ce qui prend davantage de temps encore. Quand ces données sont réparties dans les espaces

vacants de la mémoire de l'ordinateur, les listes doivent recueillir des données sur chacun de ces emplacements, ce qui ralentit encore l'accès à l'information.

Utiliser des files

Contrairement aux piles, les files sont des structures de données de type FIFO (*first in/first out*, c'est-à-dire premier entré, premier sorti). Comme pour les piles, vous pouvez trouver des applications prédéfinies dans divers modules, notamment NumPy et Pandas. Heureusement, Python vous offre aussi une application spécifique, queue, dont voici une illustration :

```
import queue

MyQueue = queue.Queue(3)

print("File vide : ", MyQueue.empty())

MyQueue.put(1)
MyQueue.put(2)
MyQueue.put(3)
print("File pleine : ", MyQueue.full())

print("Dépilage : ", MyQueue.get())
print("File pleine : ", MyQueue.full())

print("Dépilage : ", MyQueue.get())
print("Dépilage : ", MyQueue.get())
print("File vide : ", MyQueue.empty())

File vide : True
File pleine : True
Dépilage : 1
File pleine : False
Dépilage : 2
Dépilage : 3
File vide : True
```

L'utilisation du module intégré queue nécessite bien moins de code que la constitution d'une pile à partir de rien à l'aide d'une liste, mais il convient de remarquer la manière dont l'output

diffère d'une solution à l'autre. Dans l'exemple de la pile, les valeurs 1, 2 et 3 sont empilées successivement, si bien que la première valeur tirée de la pile est 3. Dans l'exemple ci-dessus, en revanche, quand on ajoute successivement à la file les valeurs 1, 2 et 3, la première valeur qui en est retirée est 1.

Se servir d'un dictionnaire

Le principe ressemble beaucoup à celui des listes, si ce n'est que vous devez maintenant définir des paires clé/valeur. Le grand avantage de cette structure de données est que les dictionnaires offrent un accès rapide à des éléments spécifiques de données, grâce à la clé. Cependant, vous ne pouvez pas utiliser n'importe quel type de clé. Il faut respecter certaines règles :

- » **La clé doit être unique.** Si vous entrez une clé qui existe déjà, elle vient simplement remplacer la définition antérieure.
- » **La clé doit être immutable.** Cela signifie que la clé peut être une chaîne, un nombre, ou encore un tuple. Mais elle ne peut pas être une liste.



La différence entre des valeurs mutables et immutables est que les valeurs immutables ne peuvent pas changer. Pour changer la valeur d'une chaîne, par exemple, Python crée en réalité une nouvelle chaîne contenant la nouvelle valeur et lui donne le nom de l'ancienne, puis supprime celle-ci.



Dans Python, les dictionnaires sont l'implémentation logicielle d'une structure de données appelée *table de hachage*, un tableau qui met les clés en correspondance avec les valeurs. Le [Chapitre 7](#) explique en détail en quoi consistent les tables de hachage et comment le hachage permet aux dictionnaires d'être plus performants en termes de rapidité. Il n'existe aucune restriction quant aux valeurs entrées. Une valeur pouvant être n'importe quel objet Python, vous pouvez vous servir d'un dictionnaire pour accéder à la fiche d'un salarié ou à d'autres

données complexes. L'exemple suivant montre comment mieux utiliser un dictionnaire :

```
Colors = {"Sam": "bleue", "Lisa": "rouge", "Sarah":
"jaune"}

print(Colors["Sarah"])
print(Colors.keys())

for Item in Colors.keys():
    print("{0} aime la couleur {1}."
        .format(Item, Colors[Item]))

Colors["Sarah"] = "pourpre"
Colors.update({"Harry": "orange"})
del Colors["Sam"]

print(Colors)

jaune
dict_keys(['Sarah', 'Lisa', 'Sam'])
Sarah aime la couleur jaune.
Lisa aime la couleur rouge.
Sam aime la couleur bleue.
{'Harry': 'orange', 'Sarah': 'pourpre', 'Lisa':
'rouge'}
```

Comme on peut le voir, un dictionnaire comporte toujours une paire clé/valeur, et la clé et la valeur sont séparées par le signe deux-points (:). Ici, pour accéder aux valeurs, vous utilisez non pas un index, mais la clé. La fonction spéciale keys vous permet d'obtenir une liste de clés que vous pouvez manipuler de différentes façons. Vous pouvez les utiliser, par exemple, pour exécuter un traitement itératif des valeurs contenues dans le dictionnaire.



Un dictionnaire est un peu comme un tableau lié à une base de données. Vous pouvez y mettre à jour, ajouter et supprimer des enregistrements, comme indiqué. La fonction update permet d'écraser des entrées ou d'ajouter de nouvelles entrées dans le dictionnaire.

Exploiter les structures arborescentes

Une structure arborescente (appelée aussi arborescence, ou arbre) a une forme analogue à celle d'un vrai arbre. Les arbres permettent d'organiser les données et de les retrouver plus rapidement qu'avec d'autres techniques de stockage. Les structures arborescentes sont communément utilisées dans les sous-programmes de recherche et de tri, mais elles ont aussi d'autres usages. Les sections qui suivent apportent un éclairage sur les arborescences de base. Dans les chapitres qui suivent, vous retrouverez des structures arborescentes dans un certain nombre d'exemples.

Aperçu des structures arborescentes

La construction d'une arborescence est à l'image d'un arbre du monde réel. Chaque élément ajouté à l'arborescence est un *nœud*. Les nœuds sont reliés les uns aux autres par des *liens*. La combinaison des nœuds et des liens donne une structure qui évoque bel et bien la forme d'un arbre ([Figure 6-1](#)).



Une arborescence possède un unique nœud racine, à l'image de l'arbre qui n'a qu'un tronc. Le nœud racine sert de point de départ aux divers traitements à exécuter. Il est relié à des branches ou à des feuilles. Un *nœud feuille* est toujours un point extrême. Les *nœuds branches* se terminent soit par de nouvelles branches, soit par des feuilles. Le type d'arborescence de la [Figure 6-1](#) est un arbre binaire, car chaque nœud possède au plus deux connexions.

Dans notre exemple, la branche B est le fils du nœud racine, celui-ci apparaissant en premier dans la liste. Les feuilles E et F sont les fils de la branche B, laquelle est donc parente des feuilles E et F. L'étude d'une arborescence consiste essentiellement à examiner les relations enfant/parent entre les nœuds. La terminologie employée est importante pour la clarté de l'étude.

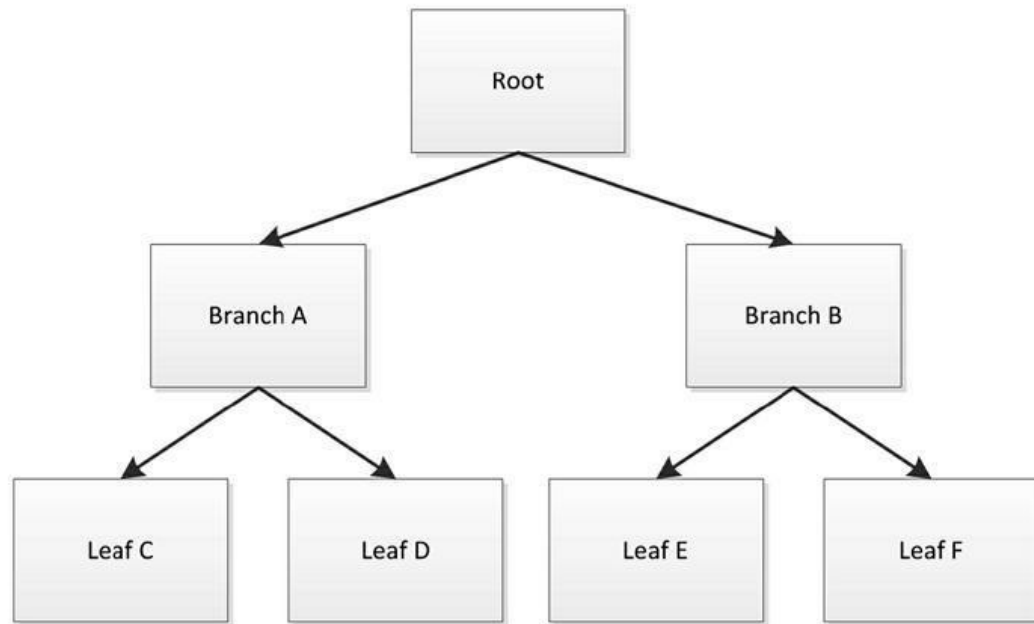


FIGURE 6-1 Dans Python, la forme d’une arborescence évoque celle d’un vrai arbre.

Créer une arborescence

Il n’y a pas d’objet arborescence tout prêt dans Python. Vous devez soit créer votre propre implémentation, soit utiliser une arborescence fournie dans un module. Pour obtenir une arborescence de base, il faut créer une classe comme support de l’objet de données. Le code suivant vous montre comment créer une classe d’arborescence de base (vous pouvez retrouver ce code dans le fichier téléchargeable A4D ; 06 ; Trees.ipynb sur le site *Dummies* : pour plus de détails, consultez l’Introduction).

```
class binaryTree:
    def __init__(self, nodeData, left=None,
right=None):
        self.nodeData = nodeData
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.nodeData)
```

Ce code crée un objet arborescence de base définissant les trois éléments qu'un nœud doit comporter : stockage de données, lien de gauche, et lien de droite. Les feuilles n'ayant pas de lien, la valeur par défaut de left (gauche) et de right (droit) est None (aucun). La classe inclut aussi une méthode pour afficher le contenu de nodeData afin de voir quelles données un nœud stocke.

Pour utiliser cette arborescence simple, vous ne devez pas essayer de stocker quoi que ce soit dans left ni dans right si ce n'est une référence à un autre nœud. Autrement, le code ne fonctionnerait pas, sachant qu'il n'y a ici aucune détection d'erreur. L'entrée nodeData peut contenir n'importe quelle valeur. Le code suivant montre comment utiliser la classe binaryTree pour construire l'arborescence de la [Figure 6-1](#) :

```
tree = binaryTree("Root")
BranchA = binaryTree("Branch A")
BranchB = binaryTree("Branch B")
tree.left = BranchA
tree.right = BranchB

LeafC = binaryTree("Leaf C")
LeafD = binaryTree("Leaf D")
LeafE = binaryTree("Leaf E")
LeafF = binaryTree("Leaf F")
BranchA.left = LeafC
BranchA.right = LeafD
BranchB.left = LeafE
BranchB.right = LeafF
```

Pour construire une arborescence, vous avez un certain nombre de possibilités, mais les deux méthodes les plus courantes consistent à le construire de haut en bas (comme dans cet exemple de code) ou de bas en haut (les feuilles en premier). Bien sûr, à ce stade, vous ne savez pas encore si l'arborescence est viable. La parcourir, c'est examiner tous les liens et vérifier qu'ils sont tous conformes à ce que vous avez prévu. Le code suivant montre comment utiliser la récursivité (voir [Chapitre 5](#)) pour parcourir cette arborescence.

```
def traverse(tree):
```

```
if tree.left != None:
    traverse(tree.left)
if tree.right != None:
    traverse(tree.right)
print(tree.nodeData)

traverse(tree)

Leaf C
Leaf D
Branch A
Leaf E
Leaf F
Branch B
Root
```

Comme le montre ce résultat, la fonction `traverse` n'imprime rien avant d'avoir atteint la première feuille. Elle imprime alors les feuilles et leurs parents. Le parcours consiste à suivre tout d'abord la branche gauche, puis la branche droite. Le nœud racine est atteint en dernier.



Il existe différents types de structures de stockage de données. Voici une brève liste des structures les plus courantes :

- » **Arbre équilibré** : Arborescence constituant une structure équilibrée grâce à une réorganisation visant à réduire les temps d'accès. Entre le nombre d'éléments du côté gauche et le nombre d'éléments du côté droit, la différence doit être au plus égale à un.
- » **Arbre non équilibré** : Arborescence dans laquelle les nouveaux éléments de données sont disposés selon les besoins sans considération d'équilibre. Avec cette méthode, la construction de l'arbre est plus rapide, mais la vitesse d'accès lors des opérations de recherche et de tri s'en trouve réduite.
- » **Tas** : Arborescence élaborée facilitant des insertions de données, ce qui rend les opérations de tri plus rapides. On peut aussi distinguer les arbres ayant en racine la clé maximale et ceux ayant en racine la clé minimale, selon la

capacité de l'arborescence à fournir immédiatement la valeur maximum ou minimum qui y est présente.

Plus loin dans ce livre, vous trouverez des algorithmes qui utilisent des arbres équilibrés, des arbres non équilibrés et des tas. Le [Chapitre 9](#), par exemple, présente l'algorithme de Dijkstra et le [Chapitre 14](#) présente le codage de Huffman. À l'aide d'illustrations et d'exemples de code, nous vous expliquons comment fonctionne chacune de ces structures de données et quel rôle elles jouent pour permettre le fonctionnement de l'algorithme.

Représenter les relations à l'aide d'un graphe

Les graphes sont aussi une forme de structure de données utilisée en algorithmique. Ils sont utilisés par exemple dans la cartographie des GPS, et dans toutes sortes de situations dans lesquelles l'approche arborescente de haut en bas ne conviendrait pas. Les sections qui suivent présentent les graphes plus en détail.

Au-delà des arborescences

Un graphe est une sorte d'extension du concept d'arborescence. On y retrouve les nœuds, souvent appelés sommets, liés les uns aux autres pour représenter les relations. Cependant, contrairement à un arbre binaire, un graphe admet plus d'une ou deux connexions à un même nœud. Dans les graphes, les nœuds, ou sommets, ont souvent une multitude de connexions. Pour rester simple, examinons le graphe de la [Figure 6-2](#).

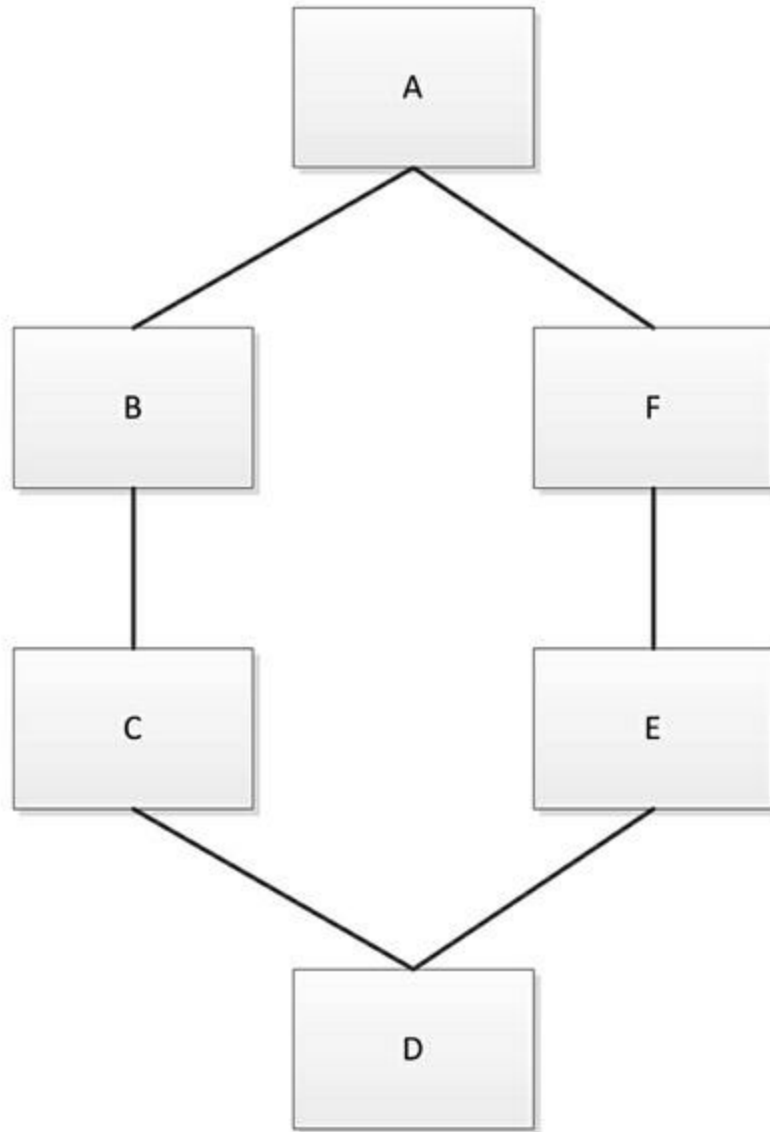


FIGURE 6-2 Les sommets d'un graphe peuvent être reliés les uns aux autres de mille manières.

Ici, le graphe forme une boucle dans laquelle A est relié à B et à F. Cependant, il n'est pas nécessaire qu'il en soit ainsi. A pourrait être un sommet isolé, comme il pourrait être relié également à C. Un graphe représente des liens entre des sommets en vue de faciliter la détermination de relations complexes.

Les graphes ajoutent aussi quelques particularités auxquelles on pourrait ne pas avoir songé. Ainsi, par exemple, un graphe peut

inclure le concept de « directionnalité ». Contrairement à un arbre, qui est constitué de relations parent/enfant, un graphe peut comporter des liens orientés dans un sens ou dans l'autre, ou non, de la même manière que dans une agglomération la plupart des rues sont bidirectionnelles, mais certaines sont à sens unique.

Dans un graphe, la représentation d'un lien ne reflète pas nécessairement toute la réalité. À certains liens, il est possible d'associer une pondération représentant par exemple la distance entre deux points, ou le temps nécessaire pour effectuer un parcours, *etc.*

Construire des graphes

Le plus souvent, pour construire des graphes, les développeurs se servent de dictionnaires (ou parfois, de listes). L'utilisation d'un dictionnaire facilite l'élaboration d'un graphe, la clé étant le nom du sommet et les valeurs correspondant aux liens de ce sommet. Voici, par exemple, le dictionnaire servant à créer le graphe de la [Figure 6-2](#) (vous retrouverez ce code dans le fichier téléchargeable A4D ; 06 ; Graphs.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
graph = {'A': ['B', 'F'],
        'B': ['A', 'C'],
        'C': ['B', 'D'],
        'D': ['C', 'E'],
        'E': ['D', 'F'],
        'F': ['E', 'A']}
```

Ce dictionnaire reflète le caractère bidirectionnel du graphe de la [Figure 6-2](#). Il pourrait tout aussi facilement définir des liens unidirectionnels ou des sommets sans aucun lien. Néanmoins, le dictionnaire est ici particulièrement bien adapté, et vous en verrez d'autres utilisations dans ce livre. Il est temps à présent de parcourir le graphe, à l'aide du code suivant :

```
def find_path(graph, start, end, path=[]):
    path = path + [start]
```

```

        if start == end:
            print("Arrivée")
            return path

    for node in graph[start]:
        print("Examen du sommet ", node)

        if node not in path:
            print("Chemin parcouru ", path)

            newp = find_path(graph, node, end, path)
            if newp:
                return newp

find_path(graph, 'B', 'E')

Examen du sommet A
Chemin parcouru ['B']
Examen du sommet B
Examen du sommet F
Chemin parcouru ['B', 'A']
Examen du sommet E
Chemin parcouru ['B', 'A', 'F']
Arrivée

['B', 'A', 'F', 'E']

```

Les chapitres qui suivent traitent de la méthode pour trouver le plus court chemin. Ici, le code ne détermine qu'un seul chemin. Il le crée sommet par sommet. Comme dans toutes les routines récursives, une stratégie de sortie est nécessaire : quand les valeurs de start et end deviennent égales, le chemin est terminé.

Sachant que chaque sommet du graphe peut être relié à plusieurs autres sommets, une boucle for est nécessaire afin de tester toutes les connexions éventuelles. Quand le sommet en question est déjà présent dans le chemin parcouru, le programme le saute. Dans le cas contraire, le programme suit le chemin et appelle de façon récursive la fonction find_path pour localiser le prochain sommet dans le graphe.

Chapitre 7

Organiser et rechercher les données

DANS CE CHAPITRE

- » Effectuer des tris à l'aide du tri fusion et du tri rapide
 - » Procéder à des recherches à l'aide d'arbres et de tas
 - » Utiliser le hachage et les dictionnaires
-

À tout moment, nous sommes entourés de données. Il n'est pas possible d'y échapper. Depuis les statistiques et autres informations qui permettent aux entreprises d'exercer leurs activités jusqu'aux recommandations nutritionnelles imprimées sur votre boîte de céréales, tout est affaire de données. Dans tous les cas, la vie des données se résume à quatre opérations de base : création, lecture, mise à jour, et suppression. En anglais, on utilise l'acronyme CRUD (*create, read, update, delete*) pour désigner ces quatre opérations axées sur la nécessité d'accéder aux données pour pouvoir exécuter toutes les tâches de l'existence facilement et rapidement. Il est donc essentiel de disposer des moyens d'organiser et de rechercher les données de différentes manières, et d'y accéder à volonté. C'est dire l'importance de ce chapitre pour quiconque désire se servir d'une application avec succès.

La première section de ce chapitre est consacrée au tri des données. Il est important de disposer les données dans un ordre qui facilite les opérations CRUD, car il est toujours préférable de pouvoir accéder aux données en utilisant le moins de code possible. En outre, même lorsque le tri des données peut sembler

secondaire, il réduit considérablement le temps de traitement dans la mesure où il est adapté à la recherche effectuée. Trier et rechercher vont de pair : on trie les données de manière à rendre la recherche plus rapide.

La deuxième section de ce chapitre est consacrée aux recherches de données. Vous ne serez pas surpris d'apprendre qu'il existe différentes manières de rechercher des données, certaines plus rapides que d'autres, certaines plus prisées des développeurs en raison de leurs caractéristiques. Le fait est qu'il n'existe aucune méthode parfaite, mais les études en vue de les améliorer se poursuivent.

La dernière section de ce chapitre est consacrée au hachage et aux dictionnaires. Le recours à l'indexation rend le tri et la recherche significativement plus rapides, mais non sans inconvénients, qu'il importe de prendre en compte (comme la consommation de davantage de ressources). Un index est en quelque sorte un pointeur, ou une adresse. Un index n'est pas une donnée, mais il pointe vers une donnée, à l'image de votre adresse qui indique où se trouve votre domicile. Imaginons qu'une personne recherche votre domicile en parcourant les pâtés de maisons l'un après l'autre et en demandant à chaque numéro si vous y habitez : cela lui prendrait énormément de temps. En recherchant votre adresse dans un annuaire de téléphone, elle pourrait vous localiser bien plus vite.

Trier les données à l'aide du tri fusion et du tri rapide

Quand on exploite des données, le tri est une des fonctionnalités les plus fondamentales. C'est pourquoi, au cours du temps, beaucoup de gens ont cherché à mettre au point de nouvelles techniques destinées à obtenir des données classées, ordonnées. Certaines de ces techniques sont plus efficaces que d'autres, et certaines conviennent particulièrement bien pour des tâches

spécifiques. Les sections qui suivent soulignent l'importance de la recherche de données et étudient les différentes possibilités d'effectuer cette recherche.

Pourquoi le tri des données est si important

On peut toujours défendre l'idée qu'il n'est pas nécessaire de trier les données. Après tout, des données qui ne sont pas triées sont tout de même accessibles, et les trier demande du temps. Naturellement, les données non triées posent le même problème qu'un tiroir rempli de bric-à-brac, dans votre cuisine ou ailleurs. Quand vous y cherchez quelque chose, cela vous prend du temps. Vous êtes obligé d'en sortir un tas d'objets avant d'apercevoir celui qui vous intéresse et de pouvoir mettre la main dessus. Seulement, l'objet dont vous avez besoin n'y est peut-être même pas : il se peut que vous l'ayez rangé ailleurs, ou que vous ne l'ayez plus.

Des données non triées dans votre système, c'est la même chose. Dès qu'une donnée est demandée, il faut examiner les données une par une, avec le risque de devoir les passer toutes en revue avant de trouver la bonne. C'est une façon plutôt frustrante de travailler avec des données. L'exemple de la recherche binaire dans la section « Diviser pour régner » du [Chapitre 5](#) met en évidence la nécessité d'un tri adéquat. Imaginez-vous en train de chercher un élément dans une liste sans l'avoir d'abord ordonnée : toute recherche serait nécessairement séquentielle et coûteuse en temps.



Naturellement, trier les données ne suffit pas. Supposons que vous utilisiez une base de données pour gérer les salariés de votre entreprise et que vous puissiez obtenir simplement un tri des salariés selon leur nom de famille. Si vous voulez procéder à une recherche en fonction de la date de naissance (par exemple, pour savoir qui fêtera son anniversaire à une date donnée), ce tri ne

vous sera pas utile. Pour faire aboutir une telle recherche, vous serez obligé de parcourir les enregistrements un à un, sur l'ensemble des enregistrements. Un tri doit donc correspondre à un besoin particulier. Pour pouvoir exploiter ces données de façon efficace, il fallait qu'elles puissent être triées par nom, mais cette fois il aurait fallu qu'elles puissent être triées par date de naissance, et une autre fois, vous aurez peut-être besoin qu'elles soient triées par service.

Le besoin de trier les mêmes données de plusieurs manières est la raison pour laquelle les développeurs créent des index. Exécuter un tri sur un petit index est plus rapide que trier l'ensemble du jeu de données. L'index permet de conserver un ordre spécifique tout en pointant sur l'ensemble du jeu de données, ce qui vous permet de trouver de façon extrêmement rapide l'information que vous voulez. En gérant un index pour chaque exigence de tri, vous pouvez réduire significativement le temps d'accès aux données et vous permettez à plusieurs utilisateurs d'y accéder en même temps et de les consulter dans l'ordre qui leur convient, en fonction de leur besoin du moment. La section « Recourir au hachage », plus loin dans ce chapitre, vous donne un aperçu du principe de l'indexation et vous montre pourquoi vous en avez parfois réellement besoin, même si la gestion de ces index consomme des ressources supplémentaires.



Les algorithmes de tri peuvent être évalués ou classés de différentes manières, notamment en fonction de la vitesse du tri. Pour mesurer l'efficacité d'un algorithme de tri en termes de rapidité, on s'intéresse généralement à deux facteurs :

- » **Les comparaisons :** Pour transférer des données d'un endroit à un autre dans le jeu de données, il faut savoir où les transférer, ce qui implique de comparer les données cibles à d'autres données du jeu de données. Or, plus les comparaisons sont nombreuses, moins le programme est performant.
- » **Les échanges :** Les données ne trouvent pas toujours leur destination finale dans le jeu de données dès la première

tentative. Tout dépend de la manière dont l'algorithme est écrit. Les données doivent parfois être déplacées plusieurs fois. Le nombre d'échanges influence considérablement la vitesse de traitement, sachant qu'en réalité on déplace les données d'une zone à une autre de la mémoire. Quand les échanges sont moins importants et moins nombreux (ce qui est le cas lorsqu'on utilise des index), le programme est plus performant.

L'algorithme de tri naïf

On appelle tri naïf le tri des données par des méthodes simples, sans réflexion concernant la manière dont les données devraient apparaître dans la liste. Par ailleurs, ces techniques consistent généralement à traiter l'ensemble du jeu de données, par opposition aux méthodes permettant de réduire le temps de tri (comme la technique de division présentée au [Chapitre 5](#)). Elles ont l'avantage d'être relativement faciles à maîtriser et d'utiliser les ressources efficacement. Par conséquent, il convient de ne pas les exclure totalement. Parmi les différentes méthodes relevant de cette catégorie, les sections suivantes en présentent les deux plus couramment utilisées.

Le tri par sélection

Le tri par sélection a remplacé le tri à bulles, qui était moins performant. Ces deux méthodes sont inefficaces, car elles s'exécutent en temps quadratique $O(n^2)$, mais le tri par sélection nécessite moins d'échanges. Un tri par sélection peut fonctionner de deux manières : soit le programme recherche le plus petit élément de la liste et le place en tête de liste, soit il recherche le plus grand élément et le place à la fin de la liste. Dans les deux cas, la méthode est très facile à mettre en application et garantit que l'élément traité apparaîtra immédiatement à sa place définitive. Voici un exemple de tri par sélection (vous le retrouverez dans le fichier téléchargeable A4D ; 07 ; Sorting

Techniques.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

for scanIndex in range(0, len(data)):
    minIndex = scanIndex
    for compIndex in range(scanIndex + 1, len(data)):
        if data[compIndex] < data[minIndex]:
            minIndex = compIndex
    if minIndex != scanIndex:
        data[scanIndex], data[minIndex] = \
            data[minIndex], data[scanIndex]
    print(data)
```

```
[1, 5, 7, 4, 2, 8, 9, 10, 6, 3]
[1, 2, 7, 4, 5, 8, 9, 10, 6, 3]
[1, 2, 3, 4, 5, 8, 9, 10, 6, 7]
[1, 2, 3, 4, 5, 6, 9, 10, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 10, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 10, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Passer au tri par insertion

Le tri par insertion consiste à utiliser un élément unique comme point de départ et à ajouter progressivement les autres éléments à gauche ou bien à droite, selon qu'ils sont plus petits ou plus grands. Chaque fois qu'un élément a été trié, l'algorithme compare un nouvel élément aux éléments triés et l'insère dans la liste, à la position qui convient. Avec cette méthode, la vitesse de tri est de type $O(n)$ dans le cas le plus favorable et de $O(n^2)$ dans le cas le plus défavorable.



Lorsque la base de données est déjà triée, nous sommes dans le cas le plus favorable, sachant que le tri par insertion ne déplacera aucune valeur. Un exemple de cas défavorable est celui d'une base de données triée dans l'ordre inverse, sachant que chaque insertion entraînera le déplacement de toutes les valeurs déjà apparues en output. Pour plus de détails sur les opérations mathématiques auxquelles ce type de tri fait appel, consultez la

page <https://www.khanacademy.org/computing/computer-science/algorithms/insertionsort/a/analysis-of-insertion-sort>.

Le tri par insertion fait encore partie des méthodes de tri par force brute, mais il nécessite moins de comparaisons que le tri par sélection. Voici un exemple de tri par insertion :

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

for scanIndex in range(1, len(data)):
    temp = data[scanIndex]
    minIndex = scanIndex

    while minIndex > 0 and temp < data[minIndex - 1]:
        data[minIndex] = data[minIndex - 1]
        minIndex -= 1

    data[minIndex] = temp
    print(data)
```

```
[5, 9, 7, 4, 2, 8, 1, 10, 6, 3]
[5, 7, 9, 4, 2, 8, 1, 10, 6, 3]
[4, 5, 7, 9, 2, 8, 1, 10, 6, 3]
[2, 4, 5, 7, 9, 8, 1, 10, 6, 3]
[2, 4, 5, 7, 8, 9, 1, 10, 6, 3]
[1, 2, 4, 5, 7, 8, 9, 10, 6, 3]
[1, 2, 4, 5, 7, 8, 9, 10, 6, 3]
[1, 2, 4, 5, 6, 7, 8, 9, 10, 3]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Opter pour de meilleures techniques de tri

Avec le progrès des technologies, les algorithmes de tri utilisent des méthodes de plus en plus subtiles. Le principe est de réduire l'ampleur du problème et de le rendre plus facile à gérer. Plutôt que de travailler sur l'intégralité du jeu de données, les algorithmes intelligents traitent des éléments de façon isolée, ce qui réduit la charge de travail nécessaire pour accomplir la tâche. Les sections suivantes étudient deux de ces techniques de tri intelligent.

Réorganiser les données avec le tri fusion

Le tri fusion consiste à appliquer l'approche « diviser pour régner ». Il s'agit d'abord de décomposer le jeu de données en parties séparées et à trier ces parties. Ensuite, il s'agit de fusionner ces parties de manière à obtenir un tri. Le tri et la fusion se poursuivent jusqu'à ce que l'ensemble du jeu de données forme à nouveau un élément unique. Le cas le plus défavorable est de type $O(n \log n)$, ce qui signifie que le tri fusion est considérablement plus rapide que les techniques présentées dans la section précédente (sachant que $\log n$ est toujours inférieur à n). Ce type de tri nécessite l'utilisation de deux fonctions. La première fonction opère de façon récursive pour séparer les parties et les remettre ensemble.

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

def mergeSort(list):
    # Détermine si la liste est divisée en
    # parties séparées.
    if len(list) < 2:
        return list

    # Trouve le milieu de la liste.
    middle = len(list)//2

    # Divise la liste en deux parties.
    left = mergeSort(list[:middle])
    right = mergeSort(list[middle:])

    # Fusionne les deux parties triées pour en faire une
    # liste unique.
    print("Côté gauche : ", left)
    print("Côté droit : ", right)
    merged = merge(left, right)
    print("Fusion ", merged)
    return merged
```

La seconde fonction fusionne les deux parties selon un processus itératif. Voici le code utilisé :

```
def merge(left, right):
    # Quand le côté gauche ou le côté droit est vide,
```

```

    # cela signifie qu'il s'agit d'un élément isolé
qui
    # est déjà trié
    if not len(left):
        return left
    if not len(right):
        return right
    # Définit les variables utilisées pour fusionner
les deux parties
    result = []
    leftIndex = 0
    rightIndex = 0
    totalLen = len(left) + len(right)

    # Continue jusqu'à ce que toutes les parties
soient réunies
    while (len(result) < totalLen):

        # Effectue les comparaisons nécessaires et
fusionne
        # les parties en fonction des valeurs
        if left[leftIndex] < right[rightIndex]:
            result.append(left[leftIndex])
            leftIndex+= 1
        else:
            result.append(right[rightIndex])
            rightIndex+= 1

        # Quand le côté gauche ou droit est plus long,
        # ajouter au résultat les éléments qui restent
        if leftIndex == len(left) or \
            rightIndex == len(right):
            result.extend(left[leftIndex:]
                          or right[rightIndex:])
            break

    return result

mergeSort(data)

```

Les instructions print dans le code vous permettent de voir comment fonctionne le processus de fusion. Ce processus peut sembler très complexe, mais en réalité il est relativement évident quand on suit les étapes de la fusion dans cet exemple.

Côté gauche : [9]


```
Côté droit : [5]
Fusion [5, 9]
Côté gauche : [4]
Côté droit : [2]
Fusion [2, 4]
Côté gauche : [7]
Côté droit : [2, 4]
Fusion [2, 4, 7]
Côté gauche : [5, 9]
Côté droit : [2, 4, 7]
Fusion [2, 4, 5, 7, 9]
Côté gauche : [8]
Côté droit : [1]
Fusion [1, 8]
Côté gauche : [6]
Côté droit : [3]
Fusion [3, 6]
Côté gauche : [10]
Côté droit : [3, 6]
Fusion [3, 6, 10]
Côté gauche : [1, 8]
Côté droit : [3, 6, 10]
Fusion [1, 3, 6, 8, 10]
Côté gauche : [2, 4, 5, 7, 9]
Côté droit : [1, 3, 6, 8, 10]
Fusion [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Résoudre les problèmes de tri de la meilleure manière par le tri rapide

Le tri rapide est, comme son nom l'indique, une des méthodes de tri les plus rapides. En consultant les articles en ligne sur le tri fusion et le tri rapide, vous constaterez que certains préfèrent l'une ou l'autre de ces deux méthodes dans une situation donnée. Ainsi, par exemple, la plupart des utilisateurs considèrent qu'un tri rapide est préférable pour trier les tableaux, tandis qu'un tri fusion convient mieux pour trier les listes liées (voir <http://www.geeksforgeeks.org/why-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists/>). Tony Hoare a écrit la première version de Quicksort en 1959, mais depuis ce temps, les développeurs ont mis au point d'autres versions du tri rapide. La

durée moyenne d'un tri rapide est $O(n \log n)$, mais dans le cas le plus défavorable, elle est $O(n^2)$.

La première partie de la tâche est la partition des données. Le programme choisit un point pivot pour définir la partie gauche et la partie droite du tri. Dans notre exemple, le code est le suivant :

```
data = [9, 5, 7, 4, 2, 8, 1, 10, 6, 3]

def partition(data, left, right):
    pivot = data[left]
    lIndex = left + 1
    rIndex = right
    while True:
        while lIndex <= rIndex and data[lIndex] <=
pivot:
            lIndex += 1
        while rIndex >= lIndex and data[rIndex] >=
pivot:
            rIndex -= 1
        if rIndex <= lIndex:
            break
        data[lIndex], data[rIndex] = \
            data[rIndex], data[lIndex]
        print(data)

    data[left], data[rIndex] = data[rIndex], data[left]
    print(data)
    return rIndex
```

LES CAS LES PLUS DÉFAVORABLES QUAND ON UTILISE LE TRI RAPIDE

Le tri rapide met rarement en jeu les cas défavorables en termes de temps de tri. Néanmoins, même les versions améliorées peuvent présenter un temps de tri défavorable $O(n^2)$ dans les situations suivantes :

- » Quand le jeu de données est déjà trié dans l'ordre voulu.
- » Quand le jeu de données est trié dans l'ordre inverse.

- » Quand tous les éléments du jeu de données sont les mêmes.

Tous ces problèmes surgissent en raison de l'utilisation d'un point pivot par une fonction de tri. Heureusement, le recours à la bonne technique de programmation permet de pallier ces problèmes en définissant comme point pivot autre chose que l'index de gauche ou l'index de droite. Les techniques utilisées dans les versions actuelles du tri rapide sont les suivantes :

- » Choisir un index aléatoire.
- » Choisir l'index médian d'une partition.
- » Choisir comme pivot la médiane de la première partie, de la partie centrale et de la dernière partie de la partition (surtout pour les partitions les plus longues).

Dans cet exemple, la boucle intérieure recherche continuellement des éléments mal placés pour les échanger. Quand le code ne peut plus échanger les éléments, il sort de la boucle et fixe un nouveau point pivot, qu'il retourne au code appelant. C'est la partie itérative du processus. La partie récursive du processus traite la partie gauche et la partie droite du jeu de données :

```
def quickSort(data, left, right):  
    if right <= left:  
        return  
    else:  
        pivot = partition(data, left, right)  
        quickSort(data, left, pivot-1)  
        quickSort(data, pivot+1, right)  
    return data  
quickSort(data, 0, len(data)-1)
```

Le nombre de comparaisons et d'échanges est relativement réduit dans cet exemple, par rapport aux autres exemples. Voici l'output de cet exemple :

```
[9, 5, 7, 4, 2, 8, 1, 3, 6, 10]  
[6, 5, 7, 4, 2, 8, 1, 3, 9, 10]  
[6, 5, 3, 4, 2, 8, 1, 7, 9, 10]
```

```
[6, 5, 3, 4, 2, 1, 8, 7, 9, 10]
[1, 5, 3, 4, 2, 6, 8, 7, 9, 10]
[1, 5, 3, 4, 2, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 8, 7, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Utiliser les arbres de recherche et le tas

Les arbres de recherche vous permettent de rechercher rapidement les données. Le [Chapitre 5](#) vous apporte les notions de recherche binaire, et la section « Exploiter les structures arborescentes » du [Chapitre 6](#) vous permet de vous familiariser davantage avec les arborescences. Obtenir les éléments de données, les placer dans le bon ordre suivant une arborescence, puis effectuer des recherches dans cette arborescence, c'est là un des moyens les plus rapides de trouver l'information.



Un cas particulier de structure arborescente est le *tas binaire*, qui consiste à placer les nœuds dans un certain ordre, le nœud racine contenant toujours la valeur la plus petite. En observant les branches de l'arbre de recherche, on constate que les branches de niveau supérieur correspondent toujours à une valeur plus petite que les branches de niveau inférieur et les feuilles. Il s'ensuit que l'arbre reste équilibré et que ses éléments sont dans un ordre prédictible, si bien que la recherche devient très efficace. Il faut simplement que l'arbre reste équilibré. Les sections suivantes présentent en détail le fonctionnement des arbres et des tas.

Se soucier de l'efficacité du processus de recherche

De toutes les tâches incluses dans les applications informatiques, la recherche est celle qui prend le plus de temps, or c'est aussi, souvent, la tâche la plus nécessaire. Même si l'addition de données (suivie de leur tri) demande un certain temps, l'intérêt de la création et de la gestion d'un jeu de données est la possibilité d'utiliser celui-ci pour effectuer des tâches utiles, ce qui suppose des recherches dans ce jeu de données en vue d'en tirer d'importantes informations. Par conséquent, on peut certes se contenter parfois d'exécuter les opérations CRUD en suivant un processus peu performant, et même, d'une routine de tri pas optimale du tout, mais les recherches doivent être aussi efficaces que possible. Le seul problème est qu'aucun algorithme de recherche n'exécute toutes les tâches de la façon la plus efficace qui soit. Il s'agit donc d'évaluer les différentes possibilités en fonction de ce que vous comptez faire avec les routines de recherche.

Deux méthodes de recherche parmi les plus efficaces consistent à utiliser l'arbre binaire de recherche (ABR) et le tas binaire. Ces deux techniques reposent sur une structure arborescente pour la gestion des clés d'accès aux éléments de données. L'organisation diffère cependant d'une méthode à l'autre, si bien que l'une ou l'autre sera préférable selon les tâches à exécuter. La [Figure 7-1](#) représente l'organisation d'un ABR.

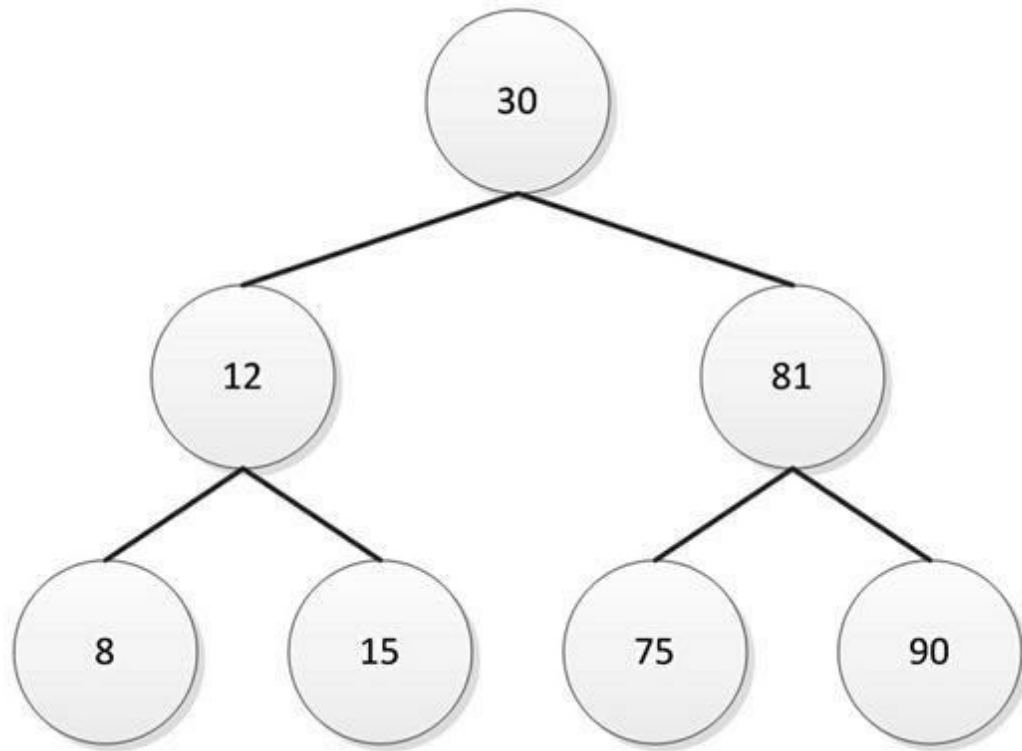


FIGURE 7-1 L'organisation des clés avec un ABR.

Il convient de remarquer qu'avec cet ordre des clés, les plus petites valeurs apparaissent à gauche et les plus grandes valeurs à droite. La valeur contenue dans le nœud racine se trouve au milieu de l'étendue des clés, si bien que cette méthode équilibrée de gestion des clés est facile à comprendre. Comparez cette organisation avec celle du tas binaire de la [Figure 7-2](#).

Sur chaque niveau, les valeurs sont inférieures à celles du niveau précédent, et la racine contient la valeur maximum parmi les clés de l'arbre. Par ailleurs, dans ce cas particulier, les plus petites valeurs apparaissent à gauche et les plus grandes à droite (même si cet ordre n'est pas strictement appliqué). Ici, il s'agit de ce que l'on appelle un *tas-max*. On pourrait aussi créer un *tas-min*, c'est-à-dire un tas dont la racine contiendrait la plus petite valeur de clé et dans lequel chaque niveau contiendrait des valeurs plus élevées que celles du niveau précédent, les valeurs les plus élevées apparaissant au niveau des feuilles.

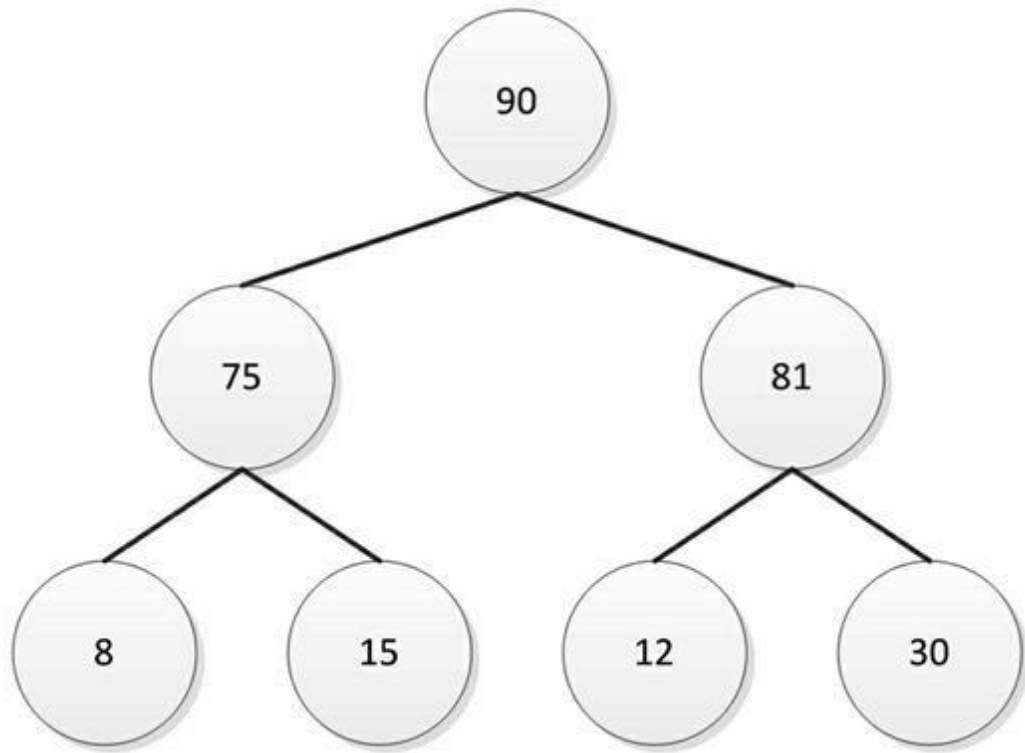


FIGURE 7-2 L'organisation des clés avec un tas binaire.



Comme noté précédemment, l'ABR présente des avantages par rapport au tas binaire lorsqu'il s'agit d'effectuer une recherche :

- » La recherche d'un élément nécessite un temps $O(\log n)$, à comparer avec un temps $O(n)$ pour un tas binaire.
- » La présentation des éléments dans l'ordre nécessite seulement un temps $O(\log n)$, à comparer avec un temps $O(n \log n)$ pour un tas binaire.
- » Trouver les valeurs extrêmes demande un temps $O(\log n)$.
- » Trouver le k -ième plus petit ou plus grand élément demande un temps $O(\log n)$ si l'arbre est correctement configuré.

L'importance de ces temps de réponse dépend de votre application. L'ABR est généralement plus performant lorsque

vous consacrez plus de temps à la recherche de valeurs qu'à la construction de l'arbre. Le tas binaire est généralement plus indiqué dans un contexte dynamique, c'est-à-dire lorsque les clés changent régulièrement. Le tas binaire a aussi ses avantages :

- » La création des structures nécessaires consomme moins de ressources, sachant que le tas binaire est lié à l'utilisation de tableaux qui se prêtent mieux au stockage en mémoire cache.
- » La création d'un tas binaire nécessite un temps $O(n)$, tandis que la création d'un ABR demande un temps $O(n \log n)$.
- » Il n'est pas nécessaire d'utiliser des pointeurs pour implémenter l'arbre.
- » L'utilisation de variantes du tas binaire (par exemple, le Tas de Fibonacci) présente des avantages comme accroître et diminuer les temps clés d'un temps $O(1)$.

Construire un arbre binaire de recherche

Diverses méthodes peuvent être utilisées pour construire un ABR. Certains se servent simplement d'un dictionnaire, d'autres d'un code personnalisé (à titre d'exemples, voir l'article de la page <https://interactivepython.org/courselib/static/pythonds/Trees/Search> ainsi que la page <http://code.activestate.com/recipes/577540-python-binary-searchtree/>). Cependant, en matière d'ABR, les développeurs n'ont généralement pas envie de réinventer la roue. C'est pourquoi il vous faut un module comme `bintrees`, qui vous offre toutes les fonctionnalités nécessaires pour créer un ABR et l'exploiter en utilisant le moins de code possible. Pour télécharger et installer `bintrees`, ouvrez une invite de commande, tapez **pip install bintrees**, et appuyez sur Entrée. La documentation de ce module se trouve sur la page <https://pypi.python.org/pypi/bintrees/2.0.6>.

Vous pouvez utiliser bintrees pour toutes sortes d'applications, mais l'exemple proposé dans cette section concerne un ABR en particulier. En l'occurrence, l'arbre est non équilibré. Le code suivant montre comment construire et afficher un ABR à l'aide de bintrees (vous le retrouverez dans le fichier téléchargeable A4D ; 07 ; Search Techniques.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
from bintrees import BinaryTree

data = {3:'Blanc', 2:'Rouge', 1:'Vert', 5:'Orange',
        4:'Jaune', 7:'Pourpre', 0:'Magenta'}

tree = BinaryTree(data)
tree.update({6:'Bleu sarcelle'})

def displayKeyValue(key, value):
    print('Clé : ', key, 'Valeur : ', value)

tree.foreach(displayKeyValue)
print('L'élément 3 contient : ', tree.get(3))
print('La valeur maximum est : ', tree.max_item())
Clé : 0 Valeur : Magenta
Clé : 1 Valeur : Vert
Clé : 2 Valeur : Rouge
Clé : 3 Valeur : Blanc
Clé : 4 Valeur : Jaune
Clé : 5 Valeur : Orange
Clé : 6 Valeur : Bleu sarcelle
Clé : 7 Valeur : Pourpre
L'élément 3 contient : Blanc
La valeur maximum est : (7, 'Pourpre')
```

Pour construire un arbre binaire, vous devez fournir des paires clé/valeur. Dans cet exemple, la méthode a consisté à créer un dictionnaire. Une fois l'arbre créé, vous pouvez vous servir de la fonction update pour ajouter de nouvelles entrées. Les entrées doivent être constituées d'une paire clé/valeur.



Cet exemple utilise une fonction pour exécuter une tâche avec les données dans tree. En l'occurrence, la fonction ne fait qu'écrire les paires clé/valeur, mais il est possible d'utiliser l'arbre comme input d'un algorithme pour l'analyse (entre autres tâches). Cette

fonction, `displayKeyValue`, sert d'input à la fonction `foreach`, laquelle affiche en output les paires clé/valeur. Vous avez également accès à mille autres fonctionnalités, comme l'utilisation de la fonction `get` pour obtenir un élément unique ou de `max_item` pour obtenir la valeur maximum stockée dans `tree`.

Effectuer des recherches dans un domaine particulier en utilisant un tas binaire

Comme c'est le cas avec un ABR, il existe un certain nombre de façons d'implémenter un tas binaire. On peut l'écrire à la main ou utiliser un dictionnaire, mais le recours à un module est plus rapide et plus fiable. Le module `heapq` étant fourni avec Python, il n'est même pas nécessaire de l'installer. Vous trouverez la documentation de ce module à l'adresse <https://docs.python.org/3/library/heapq.html>. L'exemple suivant montre comment créer et utiliser un tas binaire à l'aide de `heapq` :

```
import heapq

data = {3:'Blanc', 2:'Rouge', 1:'Vert', 5:'Orange',
4:'Jaune', 7:'Pourpre', 0:'Magenta'}

heap = []
for key, value in data.items():
    heapq.heappush(heap, (key, value))
heapq.heappush(heap, (6, 'Bleu sarcelle'))
heap.sort()

for item in heap:
    print('Key: ', item[0], 'Value: ', item[1])
print('L'élément 3 contient : ', heap[3][1])
print('Le maximum est : ', heapq.nlargest(1, heap))

Key: 0 Value: Magenta
Key: 1 Value: Vert
Key: 2 Value: Rouge
Key: 3 Value: Blanc
```

```
Key: 4 Value: Jaune  
Key: 5 Value: Orange  
Key: 6 Value: Bleu sarcelle  
Key: 7 Value: Pourpre  
L'élément 3 contient : Blanc  
Le maximum est : [(7, 'Pourpre')]
```

Cet exemple de code exécute les mêmes tâches et produit le même output que l'exemple de la section précédente, sauf qu'ici on utilise un tas binaire. Le jeu de données est le même que précédemment. Il convient cependant de noter la différence dans la façon d'ajouter des données au tas à l'aide de la fonction `heappush`. En outre, après l'addition d'un nouvel élément, il faut appeler la fonction `sort` pour que les éléments apparaissent dans le bon ordre. La manipulation de données s'apparente à la manipulation d'une liste, par opposition avec la méthode du dictionnaire utilisée avec `bintrees`. Quelle que soit la méthode utilisée, il est conseillé de faire un choix adapté à l'application que l'on veut créer et qui procure les temps de traitement les plus courts possible pour les tâches de recherche à effectuer.

Recourir au hachage

Un problème majeur que l'on rencontre avec les routines de tri est qu'elles trient les données dans un jeu de données. Or, quand ce jeu de données est de petite taille, il est difficile de se rendre compte de la quantité de données que la routine de tri doit déplacer. En revanche, quand le jeu de données est plus important, le déplacement des données devient visible : l'utilisateur doit attendre devant son écran que le traitement soit terminé. Un moyen de pallier ce problème consiste à trier uniquement les clés. La clé est la donnée qui identifie un enregistrement dans une base de données. Quand vous gérez les données relatives aux salariés d'une entreprise, le nom ou le numéro du salarié sert généralement de clé d'accès à toutes les autres informations dont vous disposez sur ce salarié. Il n'y a aucune raison de trier toutes les informations sur les salariés quand on a simplement besoin d'un tri des clés, d'où l'intérêt de recourir au hachage. Quand vous

travaillez avec ces structures de données, il est très avantageux, en termes de rapidité, de trier cette quantité de données plus réduite que représentent les clés, plutôt que l'ensemble des enregistrements.

Tout mettre dans des alvéoles

Dans tout ce qui précède, les routines de recherche et de tri consistaient à effectuer une série de comparaisons jusqu'à ce que l'algorithme trouve la valeur correcte. Or, les comparaisons ralentissent le processus, car chaque comparaison demande un certain temps.

Un moyen plus subtil d'exécuter cette tâche consiste à prédire la localisation d'un élément particulier dans la structure de données (quelle qu'elle soit) avant de le rechercher réellement. C'est ce que fait une *table de hachage* : elle permet de créer un index de clés pointant vers les différents éléments d'une structure de données de telle sorte que l'algorithme puisse facilement prédire leur localisation. Placer les clés dans l'index suppose d'utiliser une *fonction de hachage* qui transforme la clé en valeur numérique. Cette valeur numérique sert d'index dans la table de hachage, et la table de hachage fournit un pointeur vers l'enregistrement complet dans le jeu de données. Sachant que la fonction de hachage produit des résultats répétables, on peut prédire la localisation des données requises. Dans un certain nombre de cas, la table de hachage permet d'obtenir un temps de recherche $O(1)$. En d'autres termes, une seule comparaison suffit pour trouver les données.



Une table de hachage contient un nombre spécifique d'alvéoles destinées à contenir les données. Chaque alvéole peut contenir un élément de donnée. Le nombre d'alvéoles remplies rapporté au nombre d'alvéoles disponibles est le *facteur de compression*. Quand le facteur de compression est élevé, le potentiel de *collisions* (il y a collision quand deux entrées ont la même valeur de hachage) est élevé également. La section suivante de ce

chapitre explique comment éviter les collisions, mais tout ce que vous avez besoin de savoir pour le moment, c'est que des collisions peuvent se produire.

Une des méthodes les plus courantes de calcul de la valeur de hachage d'un input consiste à diviser le module de la valeur par le nombre d'alvéoles. Par exemple, si vous voulez stocker le nombre 54 dans une table de hachage comportant 15 alvéoles, la valeur de hachage sera 9. La valeur 54 ira donc dans l'alvéole n° 9 de la table, les alvéoles étant numérotées de 0 à 14. Une vraie table de hachage contient un nombre d'alvéoles considérablement plus élevé, mais le nombre 15 convient bien dans la présente section. Après avoir placé l'élément dans l'alvéole, vous pourrez utiliser une seconde fois la fonction de hachage pour le localiser.



Théoriquement, avec une fonction de hachage parfaite et un nombre infini d'alvéoles, toute valeur traitée par la fonction de hachage produira une valeur unique. Dans certains cas, le calcul du hachage peut devenir très complexe si l'on veut obtenir la plupart du temps des valeurs uniques. Cependant, plus le calcul du hachage est complexe, moins le hachage est avantageux. Le mieux est de s'en tenir à une certaine simplification.

Le hachage est compatible avec toutes sortes de structures de données. Cependant, à des fins de démonstration, l'exemple suivant utilise une liste simple de données initiales et une seconde liste constituée du résultat du hachage (vous en retrouverez le code dans le fichier téléchargeable A4D ; 07 ; Hashing.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
data = [22, 40, 102, 105, 23, 31, 6, 5]
hash_table = [None] * 15
tblLen = len(hash_table)

def hash_function(value, table_size):
    return value % table_size

for value in data:
    hash_table[hash_function(value, tblLen)] = value

print(hash_table)
```

```
[105, 31, None, None, None, 5, 6, 22, 23, None, 40,  
None,  
102, None, None]
```

Pour retrouver une valeur particulière, il suffit d'utiliser `hash_function`. Ainsi, par exemple, `print(hash_table[hash_function(102, tblLen)])` affiche 102 comme output après avoir localisé cette entrée dans `hash_table`. Les valeurs de hachage étant uniques dans ce cas particulier, `hash_function` peut à chaque fois localiser la donnée demandée.

Éviter les collisions

Un problème se pose lorsque deux entrées ont la même valeur de hachage. Si vous écrivez simplement la valeur dans la table de hachage, la seconde entrée écrasera la première, ce qui entraînera une perte de donnée. Il importe donc de pouvoir gérer les *collisions*, qui correspondent à l'utilisation d'une même valeur de hachage pour deux valeurs d'entrée. Bien sûr, la meilleure stratégie consiste à éviter toute collision.

Une méthode pour éviter les collisions consiste à s'assurer de disposer d'une table de hachage assez étendue. Maintenir un facteur de compression peu élevé est la première chose à faire pour éviter de devoir devenir créatif quand vous utilisez votre table de hachage. Cependant, même avec une table de grande dimension, on ne peut pas toujours éviter les collisions. Parfois, le jeu de données potentiel est très étendu, mais le jeu utilisé est trop réduit, et il devient impossible d'éviter ce problème. Dans un établissement scolaire où vous devez gérer 400 élèves, par exemple, si vous les identifiez par leur numéro de Sécurité sociale, les collisions sont inévitables. En effet, on ne peut pas créer une table de hachage avec un milliard d'entrées. Le gaspillage d'espace mémoire peut être considérable. La fonction de hachage devra peut-être utiliser plus qu'un simple module pour créer la valeur de hachage. Voici des techniques utilisables pour éviter les collisions :

- » **Les valeurs partielles** : Quand on gère certains types d'information, une partie de cette information se répète, ce qui peut engendrer des collisions. Ainsi, par exemple, les quatre premiers chiffres d'un numéro de téléphone pouvant être les mêmes sur une zone géographique donnée, n'utiliser que les quatre chiffres qui restent peut permettre d'éviter le problème des collisions.
- » **Le « folding »** : Créer un nombre unique peut être aussi facile que diviser le nombre initial en plusieurs fragments, rassembler les fragments et utiliser le résultat comme valeur de hachage. Prenons par exemple le numéro de téléphone 01 41 41 56 89, divisé en cinq fragments de deux chiffres. L'addition des cinq fragments donne 228, et ce nombre peut être utilisé pour produire le hachage.
- » **Le milieu du carré** : Ici, le hachage consiste à élever au carré la valeur en question et à ne retenir que les chiffres de la partie centrale du nombre obtenu, en éliminant le reste des chiffres. Considérons par exemple la valeur 120. Son carré est 14 400. On peut utiliser 440 pour produire la valeur de hachage, en éliminant le 1 à gauche et le 0 à droite.

À l'évidence, il suffit d'un peu d'imagination pour trouver toutes sortes de façons de réaliser un hachage. Malheureusement, la créativité ne permet pas de résoudre tous les problèmes de collisions : des collisions peuvent encore se produire. Il vous faut donc une autre stratégie. En cas de collision, utilisez une des méthodes suivantes pour faire face :

- » **L'adressage ouvert** : Le code parcourt séquentiellement les alvéoles jusqu'à ce qu'il en trouve une ouverte, dans laquelle il va stocker la valeur. Le problème est qu'on doit supposer qu'il existe une alvéole ouverte pour chaque valeur à traiter, ce qui n'est pas nécessairement le cas. En outre, avec l'adressage ouvert, la recherche ralentit considérablement lorsque le facteur de compression

augmente. On ne peut plus trouver la valeur cherchée dès la première comparaison.

- » **Le rehachage :** Le code hache la valeur de hachage augmentée d'une constante. Considérons par exemple la valeur 1 020, une table de hachage constituée de 30 alvéoles et une constante de 100. La valeur de hachage est 22. Si l'alvéole n° 22 contient déjà une valeur, le rehachage $((22 + 100) \% 30)$ produira une nouvelle valeur de hachage de 2. Ainsi, il n'est pas nécessaire de parcourir séquentiellement la table de hachage pour trouver une valeur. Si la méthode est appliquée correctement, un nombre limité de comparaisons doit permettre de trouver la valeur cherchée.
- » **Le chaînage :** Chaque alvéole de la table de hachage peut stocker plusieurs valeurs. On peut appliquer cette méthode en utilisant une liste à l'intérieur d'une autre liste. Chaque fois qu'une collision se produit, le programme ajoute simplement la valeur à la liste contenue dans l'alvéole cible. Cette méthode présente l'avantage d'assurer que le hachage donnera toujours l'alvéole correcte, mais dans la liste contenue par cette alvéole, une recherche séquentielle (ou autre) restera nécessaire pour trouver la valeur en question.

Créer sa propre fonction de hachage

Dans certaines situations, il peut être utile de créer des fonctions de hachage sur mesure pour répondre aux besoins de l'algorithme utilisé ou pour le rendre plus performant. Les applications cryptographiques mises à part (ce domaine mériterait qu'un ouvrage entier lui soit consacré), le [Chapitre 12](#) présente des algorithmes courants qui utilisent différentes fonctions, comme le *filtre de Bloom*, *HyperLogLog* et *Count-Min Sketch*. Ces algorithmes exploitent les propriétés des fonctions de hachage personnalisées pour extraire l'information d'énormes quantités de données.

Vous trouverez des exemples des différentes fonctions de hachage dans le module Python hashlib. Ce module comporte des algorithmes comme les suivants :

- » **Les algorithmes de hachage sécurisé (SHA) :** Il s'agit des algorithmes SHA1, SHA224, SHA256, SHA384 et SHA512. Publiés par le National Institute of Standards and Technology (NIST) en tant que standard public américain (*Federal Information Processing Standard*, ou FIPS), ces algorithmes servent des applications et des protocoles de sécurité.
- » **L'algorithme MD5 :** Initialement conçu pour des applications de sécurité, il est devenu un outil prisé pour contrôler les fichiers. Le contrôle (*checksum*) consiste à réduire le fichier à un nombre unique permettant de déterminer si ce fichier a été modifié depuis le hachage (c'est-à-dire, si le fichier téléchargé n'a pas été corrompu ni modifié par un hacker). Pour vérifier l'intégrité d'un fichier, il suffit de regarder si le checksum MD5 de votre version du fichier est le même que celui communiqué par l'auteur du fichier.



Si vous ne disposez pas de la fonction hashlib dans votre configuration Python, vous pouvez installer le module en utilisant la commande `pip install hashlib` à partir d'une commande shell. Utilisés seuls, les algorithmes de hashlib sont adaptés aux applications simples.

Néanmoins, vous pouvez combiner les résultats de plusieurs fonctions de hachage quand vous travaillez sur des applications complexes qui utilisent un grand nombre de données. Additionnez simplement les résultats des différents outputs après avoir exécuté une multiplication sur un ou plusieurs de ces résultats. La somme de deux fonctions de hachage traitées de cette manière conserve leurs propriétés même si le résultat est différent et même s'il est impossible de retrouver, à partir de ce résultat, les éléments de la somme. Cette méthode vous permet de disposer d'une fonction de

hachage vraiment nouvelle, comme une recette secrète de hachage pour vos algorithmes et vos applications.

DÉCOUVRIR DES UTILISATIONS INATTENDUES DU HACHAGE

En dehors des algorithmes présentés en détail dans ce livre, d'autres algorithmes importants sont fondés sur le hachage. L'algorithme *Locality-sensitive Hashing* (LSH), par exemple, utilise un grand nombre de fonctions de hachage pour réunir des informations qui sont séparées en apparence. Si vous vous demandez comment les sociétés de marketing et les services de renseignement rassemblent différentes bribes d'informations à partir de noms et d'adresses qui ne sont pas identiques (par exemple, lorsqu'il s'agit de comprendre que « Los Angels », « Los Angles » et « Los Angeles » font référence à Los Angeles), la réponse est LSH. LSH découpe l'information en morceaux et la digère à l'aide d'un certain nombre de fonctions de hachage, pour produire un résultat particulier qui est l'adresse d'une alvéole utilisée pour rassembler des expressions similaires. La mise en œuvre de LSH est plutôt compliquée, mais jetez un coup d'œil à cette production du Massachusetts Institute of Technology (MIT) : <http://www.mit.edu/~ando-~i/LSH/>.

L'extrait de code suivant utilise le module hashlib et les algorithmes de hachage md5 et sha1. Vous devez simplement saisir un facteur de multiplication qui sera intégré à la somme (sachant qu'il existe une infinité de nombres, votre fonction peut produire une infinité de hachages).

```
from hashlib import md5, sha1

def hash_f(element, i, length):
    """ Fonction pour créer des fonctions de hachage """
    h1 =
    int(md5(element.encode('ascii')).hexdigest(),16)
    h2 =
```

```
int(sh1(element.encode('ascii')).hexdigest(),16)
    return (h1 + i*h2) % length

print (hash_f("CAT", 1, 10**5))
64018

print (hash_f("CAT", 2, 10**5))
43738
```

Si vous vous demandez où trouver autour de vous d'autres utilisations des tables de hachage, intéressez-vous aux dictionnaires de Python. Les dictionnaires sont, en réalité, des tables de hachage, même si leur façon de gérer les collisions est subtile si et vous ne perdrez pas vos données lorsque deux clés hachées donnent le même résultat. Le fait que l'index du dictionnaire utilise un hachage est aussi la raison de la rapidité avec laquelle il vérifie la présence de la clé. Par ailleurs, l'utilisation d'un hachage explique pourquoi on ne peut pas utiliser n'importe quel type de donnée comme clé. La clé que vous choisissez doit être quelque chose que Python pourra transformer en résultat d'un hachage. Les listes, par exemple, ne peuvent pas être hachées car elles sont mutables : il est possible de les modifier en ajoutant ou en retirant des éléments. Néanmoins, si vous transformez votre liste en chaîne, vous pourrez l'utiliser comme clé pour un dictionnaire dans Python.

PARTIE 3

Explorer le monde des graphes

DANS CETTE PARTIE...

Assimiler les bases pour pouvoir tracer, mesurer et analyser les graphes

Utiliser des graphes pour localiser les sommets, trier les éléments et trouver le plus court chemin

Se servir de graphes pour étudier les réseaux sociaux

Étudier les graphes pour dégager des tendances et prendre des décisions en conséquence

Utiliser l'algorithme PageRank pour classer les pages Web

Chapitre 8

Assimiler les bases de la théorie des graphes

DANS CE CHAPITRE

- » Apprécier l'importance des réseaux
 - » Découvrir les techniques de traçage de graphes
 - » Étudier les fonctionnalités des graphes
 - » Utiliser des formats numériques pour représenter des graphes
-

Les *graphes* sont des structures constituées d'un certain nombre de sommets (ou nœuds) reliés entre eux par des arêtes ou par des arcs (selon la manière dont on les représente). Un graphe est à l'image d'un plan de rues sur lequel les intersections seraient les sommets, et les rues seraient les arêtes. Cette présentation diffère de l'arborescence dans laquelle tous les chemins se terminent sur un nœud feuille. Au [Chapitre 7](#), nous avons vu qu'une arborescence pouvait ressembler à un organigramme ou à un arbre généalogique. Plus important, les structures arborescentes ressemblent véritablement à des arbres et présentent bien un point de départ et un point d'arrivée. Dans ce chapitre, il s'agit tout d'abord de bien comprendre l'importance des réseaux, qui sont des graphes communément utilisés dans de nombreux domaines.



Il existe toutes sortes de façons de représenter un graphe, la plupart du temps de façon abstraite. À moins que vous ayez une aptitude particulière à visualiser les choses mentalement (ce qui

n'est pas le cas de la plupart d'entre nous), il importe que vous sachiez tracer un graphe afin de le voir réellement. C'est généralement la vision des choses qui nous permet de comprendre leur fonctionnement. Un langage comme Python est très bien adapté au traçage de graphes, qui est une des fonctionnalités essentielles. C'est même une des raisons pour lesquelles nous avons choisi d'utiliser Python dans ce livre plutôt qu'un autre langage comme le C (qui conviendrait très bien pour des tâches très différentes).

Après la visualisation d'un graphe, il est important de savoir comment exploiter une représentation graphique. Ce chapitre commence par mesurer les fonctionnalités des graphes. Pour déterminer la complexité et d'autres caractéristiques d'un graphe, on compte notamment les arêtes et les sommets. La visualisation des graphes facilite certaines tâches comme la détermination de la centralité. Naturellement, les éléments exposés dans ce chapitre seront exploités au [Chapitre 9](#).

La présentation numérique d'un graphe est une chose importante, même si elle ne permet pas une compréhension aisée du graphe. La présentation graphique est réalisée à votre attention, mais elle n'est pas compréhensible pour l'ordinateur (même s'il l'a réalisée). Tout se passe comme si la machine avait besoin d'abstraction. S'agissant de mettre les graphes sous une forme compatible avec le fonctionnement de l'ordinateur, ce chapitre étudie trois techniques de transcription d'un graphe en format numérique : les matrices, les représentations éparses, et les listes. Chacune de ces techniques a ses avantages et ses inconvénients, et elles feront l'objet d'utilisations spécifiques dans les chapitres ultérieurs (à partir du [Chapitre 9](#)). Il en existe d'autres, mais ces trois méthodes feront bien l'affaire pour communiquer avec l'ordinateur.

Apprécier l'importance des réseaux

Un *réseau* est un type de graphe associant des noms à des sommets (ou nœuds) et/ou à des arêtes (ou arcs, ou lignes). L'utilisation de noms rend la représentation graphique moins abstraite et en facilite la compréhension. Le lecteur a ainsi une vision plus concrète des données représentées, même si le graphe reste une représentation abstraite de la réalité sous une forme pouvant être comprise de façon différente par l'utilisateur et par la machine. Les sections qui suivent vous permettent de mieux apprécier l'importance des réseaux et de vous rendre compte de la manière dont leur utilisation dans ce livre simplifie la compréhension des algorithmes et des avantages que vous pouvez en tirer.

Ce que recouvre la notion de graphe

Un graphe est constitué de paires ordonnées sous la forme $G = (V, E)$, où G est le graphe, V est une liste de vertex, c'est-à-dire de sommets, et E une liste d'arêtes reliant les sommets. Une arête est en réalité une paire numérique représentant une liaison entre deux sommets. Supposons que les sommets représentent des villes, et que vous vouliez relier Le Mans (dont la valeur est 1) à Alençon (dont la valeur est 2). Vous allez créer une arête appelée Autoroute, à laquelle sera associée une paire de références : $\text{Autoroute} = [\text{Le Mans}, \text{Alençon}]$. Le graphe sera noté $G = [(\text{Le Mans}, \text{Alençon})]$, ce qui signifie simplement qu'un premier sommet nommé Le Mans est relié à un second sommet nommé Alençon. En utilisant l'ordre de présentation des sommets, Le Mans est adjacent à Alençon : en d'autres termes, une voiture qui part du Mans arrivera à Alençon.

Les graphes peuvent avoir plusieurs formes. Un *graphe non orienté* (comme celui de la [Figure 8-1](#)) est un graphe dans lequel l'ordre des entrées n'a pas d'importance. Une carte routière est généralement assimilable à un graphe non orienté, puisque la circulation sur chaque route peut se faire dans les deux sens.

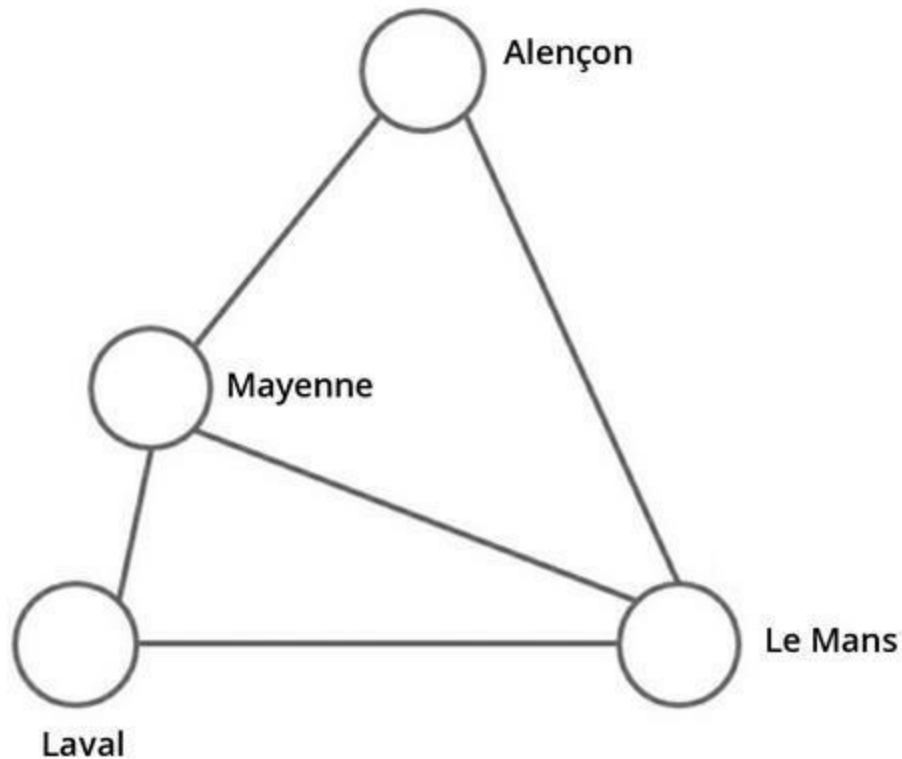


FIGURE 8-1 Un graphe simple non orienté.

Un *graphe orienté*, comme celui de la [Figure 8-2](#), est un graphe dans lequel l'ordre des entrées est important car le flux transite de la première entrée vers la deuxième. Dans ce cas, on parle d'arcs plutôt que d'arêtes. Prenons l'exemple de la représentation graphique d'une séquence de signalisation lumineuse, avec Rouge = 1, Orange = 2 et Vert = 3. Les trois arcs nécessaires pour représenter cette séquence sont : Passez = [Rouge, Vert], Attention = [Vert, Orange], et Stop = [Orange, Rouge]. L'ordre des entrées est important : si le système de signalisation ignorait l'ordre de cette séquence, il n'est pas difficile d'imaginer le chaos qui en résulterait.

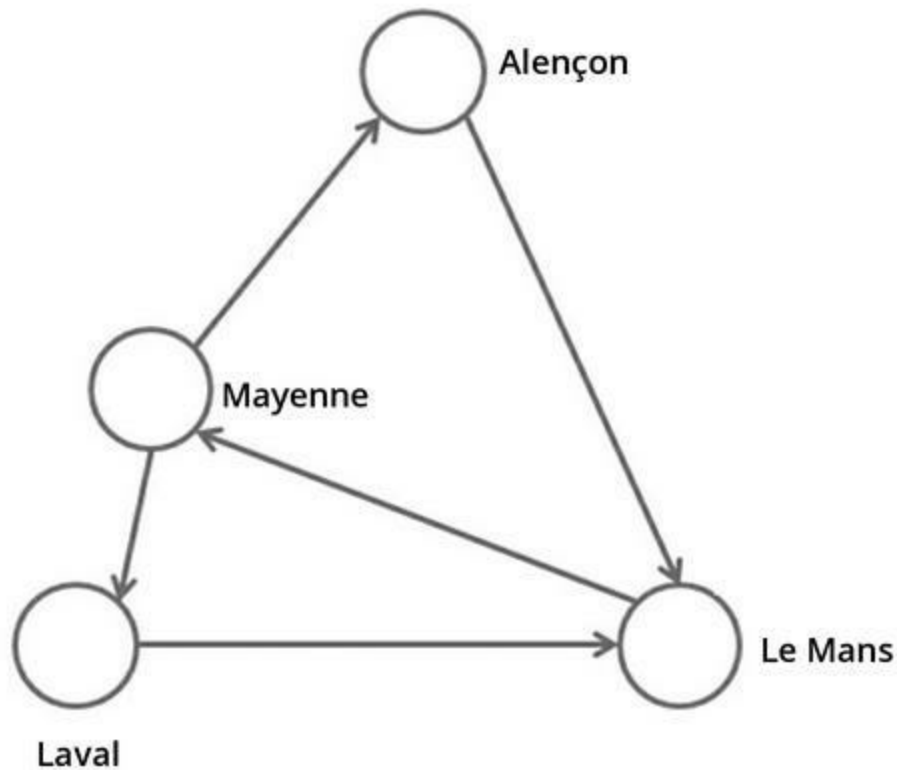


FIGURE 8-2 La version orientée du même graphe.

Un troisième type de graphe essentiel à étudier est le graphe partiellement orienté. Reprenons l'exemple de la carte routière. La circulation n'est pas nécessairement à double sens sur toutes les routes. Sur certaines cartes, il importe de tenir compte des voies à sens unique, surtout en agglomération. Par conséquent, dans un même graphe, on peut avoir besoin à la fois d'un sous-graphe non orienté et d'un sous-graphe orienté : l'ensemble est ce que l'on appelle un *graphe partiellement orienté*.

Un autre type de graphe à connaître est le *graphe pondéré* (Figure 8-3) : c'est un graphe avec des valeurs assignées aux arêtes ou aux arcs. Reprenons notre exemple de la carte routière. La direction à prendre n'est souvent pas la seule information désirée : l'utilisateur peut vouloir connaître la distance ou le temps de trajet jusqu'à la prochaine destination. C'est le type d'information que donne un graphe pondéré. Les valeurs (ou

poids) peuvent être utilisées de différentes manières dans les calculs associés aux graphes.

Si l'on assimile une carte routière à un graphe, non seulement des valeurs sont affectées aux arêtes, mais des noms sont affectés aux sommets : autrement, l'utilisateur verrait les villes, mais aurait des difficultés à les identifier. D'autres types de graphes sont présentés sur la page <http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html>.

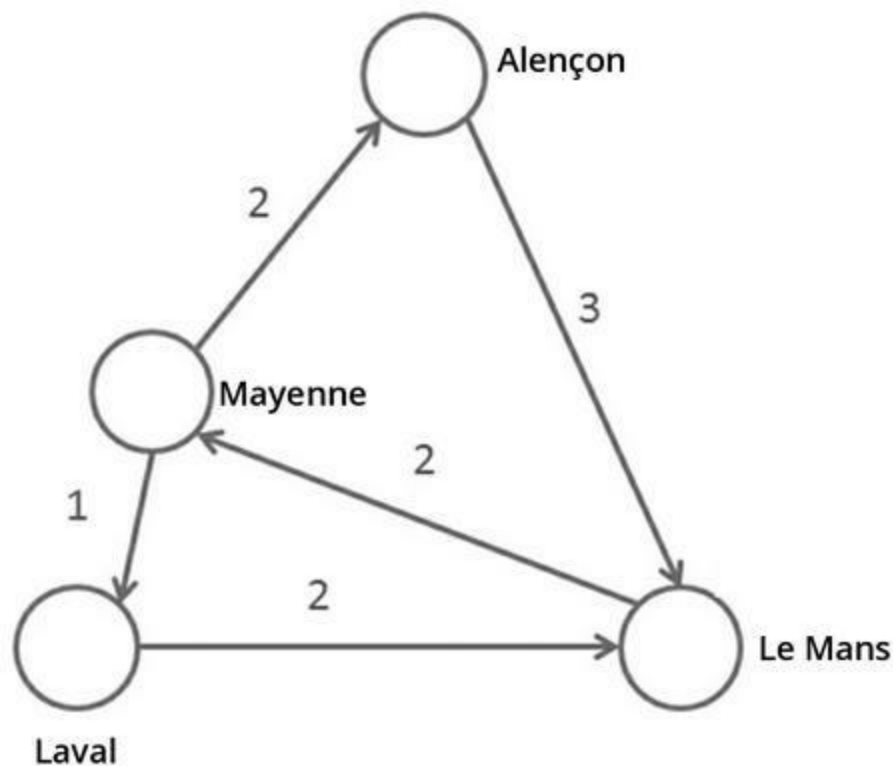


FIGURE 8-3 Un graphe pondéré pour plus de réalisme.

Quand on retrouve partout des graphes

Même s'ils vous rappellent certains problèmes de mathématiques que vous trouviez abstraits ou ennuyeux au cours de vos études,

les graphes, en réalité, sont un domaine très intéressant à étudier, car nous nous en servons tous, à tout moment, sans même nous en rendre compte. Naturellement, la plupart du temps vous ne vous souciez pas de tous ces nombres. Une carte routière est un graphe, même si elle apparaît comme quelque chose de plus parlant et de plus concret avec des villes, des routes et toutes sortes d'autres éléments. Pour vous, en pratique, c'est une carte et non un graphe, mais du point de vue de votre GPS c'est bel et bien un graphe, et c'est ce qui lui permet de toujours vous proposer l'itinéraire le plus court vers votre destination. En cherchant un peu, vous pourriez vous rendre compte que vous êtes entouré de graphes, même si vous n'avez pas l'habitude de les voir comme tels.



Certains graphes ne sont pas typiquement visibles, et l'on ne se rend souvent pas compte que ce sont des graphes. Les systèmes de menu téléphonique, par exemple, sont une forme de graphe orienté, et malgré leur apparente simplicité, ils sont plutôt compliqués. Ils peuvent comporter des boucles et toutes sortes d'autres structures. Vous pourriez vous lancer dans un exercice intéressant, consistant à représenter le graphe d'un système de menu : vous seriez sans doute surpris de vous apercevoir à quel point le problème peut être compliqué.

On retrouve aussi des systèmes de menu dans les applications informatiques. Pour pouvoir exécuter des tâches, les applications interactives procèdent généralement selon une série d'étapes au cours desquelles elles font appel à une sous-application d'un type particulier, appelée un assistant. Grâce aux assistants, des applications qui peuvent paraître compliquées deviennent bien plus faciles à utiliser. Cependant, pour que ces assistants fonctionnent, le développeur doit mettre au point un graphe décrivant la série d'étapes.

Aussi surprenant que cela puisse paraître, même les recettes de cuisine peuvent être des graphes (et la représentation graphique des relations entre les ingrédients peut se révéler intéressante). Dans une recette, chaque ingrédient est un sommet. Les arêtes qui relient ces sommets sont les instructions qui concernent le mélange des ingrédients. Bien sûr, une recette de cuisine relève

davantage de la chimie que des mathématiques, mais en chimie aussi, on utilise des graphes pour représenter la relation entre les éléments qui constituent une molécule (à propos des recettes de cuisine considérées comme des graphes, consultez la page <http://stackoverflow.com/questions/7749073/representing-a-cooking-recipe-in-a-graph-database>).



En résumé, nous sommes tout le temps confrontés à des graphes, même si nous ne les voyons pas comme tels : recettes de cuisine, formules chimiques, *etc.* Les graphes représentent toutes sortes de relations entre des objets, partout où existe une séquence ordonnée, une dépendance temporelle ou une causalité.

Quand les graphes ont un aspect social

Les graphes peuvent avoir des implications sociales, sachant qu'ils servent souvent à refléter des relations entre des personnes, dans des contextes variés. Un des domaines d'application les plus évidents est l'organigramme d'une organisation. Chaque sommet est une personne, et les arcs représentent les liens hiérarchiques. Il en est de même de toutes sortes de graphes comme par exemple la représentation d'un arbre généalogique. L'organigramme est cependant un graphe non orienté, car la communication circule dans les deux sens entre un supérieur hiérarchique et son subordonné (même si elle revêt un aspect différent selon le sens dans lequel elle se fait). En revanche, l'arbre généalogique est un graphe orienté, car chaque enfant a deux parents. Le flux représente le sens de l'hérédité, depuis les ancêtres les plus anciens jusqu'aux enfants actuels.

L'utilisation des graphes est utile également aux réseaux sociaux. Ainsi, par exemple, l'analyse des liens entre les tweets sur Twitter fait l'objet aujourd'hui de toute une industrie (pour un exemple, voir <http://twittertoolsbook.com/10-awesome-twitter-analytics->

[visualizationtools/](#)). Cette analyse repose sur l'utilisation de graphes.

Cependant, pour trouver des exemples de graphes utilisés, il n'est pas nécessaire de chercher ailleurs que dans le courrier électronique. La base de données Enron corpus contient 200 399 messages électroniques envoyés par 158 cadres dirigeants, et ces messages ont été diffusés sur Internet par la Federal Energy Regulatory Commission (FERC). En 2001, des scientifiques et des universitaires s'en sont servis pour élaborer des graphiques sociaux afin de faire savoir que la septième plus grande société américaine devait déposer son bilan (pour savoir comment ces informations ont permis et permettent encore de progresser dans l'analyse de graphes complexes, voir <https://www.technologyreview.com/s/515801/the-immortal-life-of-the-enron-e-mails/>).

Les graphes à finalité sociale, vous pouvez en trouver même dans votre ordinateur. Quelle que soit votre application de messagerie, vous pouvez grouper vos courriels de diverses manières et ces regroupements font appel aux graphes. Si ce n'était pas le cas, vous ne sauriez pas que tel message est une réponse à tel autre message. Plus le nombre de messages augmenterait et plus il vous faudrait d'efforts et de temps pour parvenir à vous y retrouver.

Comprendre les sous-graphes

Les relations représentées par les graphes peuvent devenir très complexes. Sur les plans de rues, par exemple, la plupart des voies de circulation sont à double sens, si bien qu'un graphe non orienté semble être le mode de représentation le plus approprié. Cependant, un certain nombre de rues sont à sens unique, ce qui implique l'utilisation d'un graphe orienté. La présence de voies à double sens et à sens unique rend impossible (ou du moins, difficile) une représentation à l'aide d'un unique type de graphe. Pour pouvoir réunir des graphes orientés et non orientés sous forme d'un graphe unique, il faut donc créer des sous-graphes,

puis les relier les uns aux autres. On parle alors de graphes partiellement orientés.

Les sous-graphes servent d'autres fins également. Pour étudier une boucle dans un graphe, par exemple, il est utile de la considérer comme un graphe élémentaire et de ne prendre en compte que les sommets et les arêtes concernés. Cette méthode sert dans toutes sortes de disciplines. Les informaticiens, notamment, s'en servent pour vérifier que telle ou telle partie d'une application fonctionne comme prévu, et les urbanistes s'en servent aussi pour analyser les problèmes de circulation dans les secteurs les plus denses d'une agglomération. Dans le secteur médical, ce sont aussi les sous-graphes qui permettent d'étudier la circulation du sang ou d'autres liquides entre les organes du corps humain. Les organes sont les sommets et les vaisseaux sanguins sont les arcs. Bien souvent, ces graphes sont pondérés : il peut être essentiel, par exemple, de savoir non seulement par où le sang circule, mais également quelle quantité de sang circule.

Des graphes complexes peuvent aussi receler des tendances qu'il importe de découvrir. Ainsi, par exemple, un même cycle peut se retrouver en divers endroits du graphe, ou dans différents graphes. L'élaboration d'un sous-graphe en fonction de ce cycle facilite les comparaisons à l'intérieur d'un même graphe, ou entre les graphes. Un biologiste, par exemple, peut vouloir comparer le cycle de mutation d'une espèce animale à celui d'une autre espèce. Pour ce faire, il décrira le processus à l'aide d'un sous-graphe. Pour un exemple intéressant, voir la page <http://www.sciencedirect.com/science/article/pii/S1359027896000> Le graphe apparaît sur la Figure 1, vers le début de l'article.

Comment tracer un graphe

Rares sont les gens capables de se représenter les données mentalement. Pour la plupart d'entre nous, nous avons besoin d'une représentation graphique. C'est ce qu'illustre l'utilisation de graphiques dans les présentations organisées par les entreprises. Si

vous présentez à votre audience les résultats des ventes de l'année écoulée uniquement par des tableaux chiffrés, vos interlocuteurs ne tarderont pas à se déconcentrer, et vous ne ferez pas passer votre message. Certes, les tableaux permettent de présenter un grand nombre d'informations de façon précise, mais ce n'est pas la forme de communication la plus accessible.



En représentant graphiquement les données et en montrant les chiffres de ventes sous forme de diagrammes en barres, vous facilitez grandement la compréhension par votre audience des relations entre les nombres. Si les ventes sont en augmentation d'année en année, la longueur croissante des barres le montrera clairement. Il est cependant intéressant de remarquer que la représentation graphique est moins précise. En observant un diagramme, il est pratiquement impossible de dire que l'entreprise a réalisé un chiffre d'affaires de 3 400 026,15 € l'année dernière et de 3 552 215,82 € cette année. Cette information serait visible sur le tableau, mais en réalité personne n'a besoin de connaître ce niveau de détail : ce qui importe, c'est l'accroissement annuel et l'appréciation de la différence entre deux années. L'ordinateur, lui, aime les détails, c'est pourquoi les graphiques s'adressent aux humains et les matrices aux ordinateurs.

Les sections suivantes vous présentent les merveilles de la représentation graphique. Elles vous donnent un aperçu rapide de la manière dont on utilise les graphes avec Python. Naturellement, ces principes seront présentés plus en détail dans les chapitres qui suivent. Ces sections vous donnent les bases pour pouvoir plus facilement comprendre les graphes qui seront présentés ensuite.

Les principales caractéristiques des graphes

Avant de pouvoir tracer un graphe, vous avez besoin d'acquérir des notions concernant les caractéristiques des graphes. Comme mentionné précédemment, tout graphe est constitué de sommets

(ou nœuds) et d'arêtes (pour les graphes non orientés) ou d'arcs (pour les graphes orientés). Cependant, la représentation de ces éléments dépend en partie du module utilisé. Par souci de simplification, ce livre utilise deux modules à la fois :

- » **NetworkX** (<https://networkx.github.io/>) : Contient du code pour tracer des graphes.
- » **matplotlib** (<http://matplotlib.org/>) : Donne accès à toutes sortes de routines de traçage, dont certaines peuvent afficher des graphes créés avec NetworkX.



Pour pouvoir utiliser ces modules dans Python, vous devez les importer. L'utilisation de modules externes implique l'ajout d'un code spécial, comme les lignes de code suivantes qui donnent accès à matplotlib et à networkx (vous le retrouverez dans le fichier téléchargeable A4D ; 08 ; Draw Graph.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
```



L'entrée spéciale `%matplotlib inline` vous permet de voir vos graphes directement dans le notebook plutôt que sous forme de représentation graphique externe. Vous pouvez ainsi créer un notebook avec des graphes déjà inclus et vous n'avez donc pas besoin d'exécuter à nouveau le code pour voir les résultats obtenus précédemment.

Maintenant que vous avez accès aux modules, vous pouvez créer un graphe. Ici, le graphe est une sorte de boîte regroupant les principales caractéristiques qui le définissent. Ainsi, vous pouvez tracer le graphe pour pouvoir l'étudier par la suite. Le code suivant crée l'objet NetworkX Graph.

```
AGraph = nx.Graph()
```

Ensuite, les principaux attributs sont ajoutés à AGraph. Vous devez ajouter les sommets et les arêtes en utilisant le code

suivant :

```
Nodes = range(1,5)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5)]
```

Comme mentionné précédemment, Edges décrit les connexions entre les Nodes (sommets). En l'occurrence, Nodes contient les valeurs 1 à 5, et Edges contient donc les connexions entre ces valeurs.

Naturellement, les Nodes (sommets) et les Edges (arêtes) n'apparaîtront pas dans AGraph. Si vous voulez les voir, vous devez les mettre dans la boîte. Pour ajouter à AGraph les Nodes et les Edges, utilisez le code suivant :

```
AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)
```

Le module NetworkX contient toutes sortes de fonctions que vous pouvez utiliser pour interagir avec les sommets et les arêtes, mais la méthode montrée ici est la plus rapide. Et cependant, peut-être devrez-vous par la suite ajouter des sommets. Vous pourriez, par exemple, envisager d'ajouter un sommet entre 2 et 4, auquel cas vous appelleriez la fonction AGraph.add_edge(2, 4).

Tracer le graphe

Vous pouvez interagir de toutes sortes de façons avec l'objet boîte AGraph créé dans la section précédente, mais si vous êtes du genre visuel, vous risquez de trouver un certain nombre de ces méthodes abstraites et peu satisfaisantes. Pour voir ce que contient un objet, le mieux est souvent de l'observer. Le code suivant affiche le graphe contenu dans AGraph :

```
nx.draw(AGraph, with_labels=True)
```

La fonction draw() fournit divers arguments utilisables pour enjoliver le graphe. Vous pouvez, par exemple, changer la couleur

des sommets en utilisant l'argument `node_color` et la couleur des arêtes en utilisant l'argument `edge_color`. La [Figure 8-4](#) représente le graphe contenu dans `AGraph`.

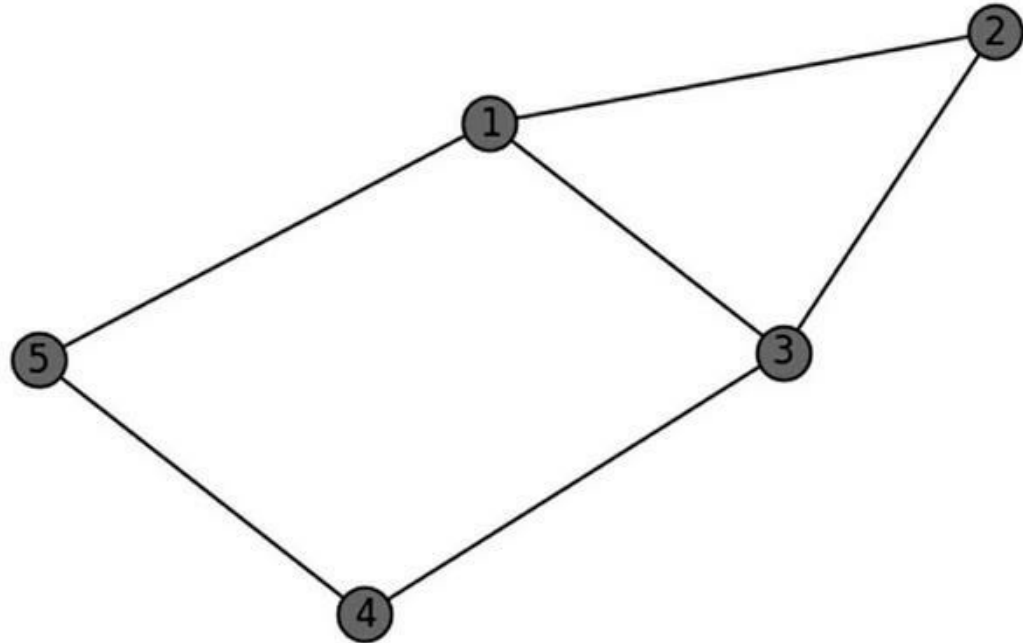


FIGURE 8-4 Il est plus facile de comprendre un graphe quand on voit ce qu'il contient.

Mesurer la fonctionnalité d'un graphe

Une fois que vous êtes en mesure de visualiser et de comprendre un graphe, vous devez vous demander quelles parties du graphe sont importantes. En effet, mieux vaut ne pas perdre de temps à effectuer une analyse sur des données qui ne jouent pas un grand rôle dans l'histoire. Prenons l'exemple d'une analyse du trafic automobile effectuée en vue d'améliorer le réseau de rues. Les intersections sont les sommets et les rues sont les arêtes le long desquelles se fait la circulation. Étudier comment cette circulation se répartit, c'est-à-dire par quels sommets et par quelles arêtes

passer le plus gros trafic, permet de savoir quelles voies pourraient être utilement élargies et quelles voies auront besoin d'un entretien plus fréquent, compte tenu d'une usure plus rapide du revêtement.

Cependant, étudier le cas de chacune des voies ne saurait suffire. Il se peut qu'un nouveau gratte-ciel attire un trafic important, avec un impact sur toute une zone. Cet édifice est alors un point central autour duquel la circulation se densifie. Les sommets les plus importants sont les plus proches du gratte-ciel. La détermination de la *centralité*, c'est-à-dire des sommets les plus importants du graphe, vous permet de savoir quelles parties exigent la plus grande attention. Les sections suivantes sont consacrées aux questions fondamentales que vous devez aborder quand vous mesurez la *fonctionnalité d'un graphe*, qui est sa capacité à modéliser un problème donné.

Compter les arêtes et les sommets

Quand les graphes deviennent plus complexes, ils fournissent plus d'informations, mais ils deviennent aussi plus difficiles à comprendre et à manipuler. Le nombre d'arêtes et de sommets dans un graphe est ce qui détermine sa complexité. Cependant, tout dépend de la combinaison de ces arêtes et de ces sommets. Il se peut, par exemple, qu'un sommet ne soit relié à aucun autre sommet. Faire figurer un tel sommet dans un graphe se justifie s'il s'agit de représenter une valeur qui n'a pas de lien avec les autres valeurs. À l'aide du code suivant, vous pouvez facilement déterminer que le sommet 6 n'est pas relié aux autres sommets car il n'y a pas, pour ce sommet, de données relatives à des arêtes (vous retrouverez ce code dans le fichier A4D ; 08 ; Graph Measurements.ipynb).

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
```

```
AGraph = nx.Graph()
```

```
Nodes = range(1,5)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

AGraph.add_node(6)
sorted(nx.connected_components(AGraph))

[{'1', '2', '3', '4', '5'}, {'6'}]
```

DES DIFFÉRENCES AU NIVEAU GRAPHIQUE

La [Figure 8-4](#) représente un output type. Il se peut que votre graphe soit légèrement différent. Le triangle, par exemple, peut apparaître dans la partie inférieure du graphe plutôt que dans la partie supérieure, ou bien les angles entre les sommets peuvent varier. Les connexions entre les sommets sont l'aspect le plus important, et de légères différences au niveau de l'apparence peuvent être négligées. Si vous exécutez le code plusieurs fois, vous constaterez que l'orientation du graphe change, ainsi que les angles entre les sommets. La même différence s'observe sur d'autres captures d'écran dans ce livre. Quand vous observez un graphe, intéressez-vous toujours aux connexions entre les sommets plutôt que de vous attendre à une correspondance exacte entre votre output et celui du livre.

L'output de ce code montre que les sommets 1 à 5 sont reliés et que le sommet 6 n'est relié à rien. Vous pouvez, bien sûr, remédier à cette situation en ajoutant une autre arête, à l'aide du code suivant, après quoi vous vérifierez le résultat :

```
AGraph.add_edge(1,6)
sorted(nx.connected_components(AGraph))

[{'1', '2', '3', '4', '5', '6'}]
```

L'output montre maintenant que chaque sommet est relié à au moins un autre sommet. Cependant, vous ignorez quels sommets

ont le plus de connexions. Le nombre d'arêtes d'un sommet s'appelle le *degré* du sommet. Plus ce degré est élevé, plus le sommet devient complexe. En fonction de leur degré, on a une idée de l'importance des sommets. Le code suivant montre comment obtenir le degré dans le graphe qui nous sert d'exemple :

```
nx.degree(AGraph).values()  
dict_values([4, 2, 3, 2, 2, 1])
```

Les degrés apparaissent dans l'ordre des sommets : le sommet 1 possède quatre connexions et le sommet 6 n'a qu'une connexion. Le sommet 1 est donc le plus important, suivi par le sommet 3 qui a trois connexions.

Dans la modélisation de données réelles comme les tweets sur un sujet particulier, les sommets ont aussi tendance à s'associer. On pourrait y voir une tendance actuelle. En mathématiques, on parle de regroupement, d'agrégation ou de *clustering*, et la mesure de cette tendance permet de savoir quel est, dans un graphe, le groupe de sommets le plus important. Voici le code à utiliser pour mesurer ce que l'on appelle le coefficient de clustering, dans notre exemple :

```
nx.clustering(AGraph)  
  
{1: 0.16666666666666666, 2: 1.0, 3:  
0.3333333333333333,  
4: 0.0, 5: 0.0, 6: 0.0}
```

Cet output montre que les sommets se regroupent plutôt autour du sommet 2, même si c'est le sommet 1 qui présente le plus haut degré. En effet, les sommets 1 et 3 présentent des degrés élevés et le sommet 2 se trouve entre l'un et l'autre.



Le *clustering* permet de mieux analyser les données. Cette technique permet de constater que certains sommets du graphe sont mieux reliés, tandis que d'autres risquent de se retrouver isolés.

Connaître la façon dont les éléments sont reliés dans un graphe permet de déterminer les possibilités d'en renforcer la structure, ou au contraire, de la défaire. Durant la Guerre froide, les experts militaires américains et soviétiques étudiaient le regroupement pour savoir comment il serait possible d'interrompre la chaîne d'approvisionnement de l'adversaire en cas de conflit.

Calculer la centralité

La centralité peut prendre différentes formes, sachant que l'importance des éléments dépend souvent de plusieurs facteurs. Les éléments importants d'un graphe ne seront pas les mêmes selon qu'il s'agira d'analyser des tweets ou d'étudier le flux de circulation automobile. Heureusement, NetworkX vous propose différentes méthodes de calcul de la centralité. Vous pouvez la calculer, par exemple, en fonction des degrés des sommets. Le code suivant utilise le graphe modifié de la section précédente de ce chapitre (vous retrouverez ce code dans le fichier A4D ; 08 ; Graph Centrality.ipynb).

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

AGraph = nx.Graph()

Nodes = range(1,6)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5),
(1,6)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

nx.degree_centrality(AGraph)

{1: 0.8, 2: 0.4, 3: 0.6000000000000001, 4: 0.4, 5:
0.4,
6: 0.2}
```

L'UTILISATION DU SAUT DE LIGNE DANS L'OUTPUT

Dans cet exemple, l'output est sur deux lignes, alors qu'il apparaît sur une seule ligne dans Jupyter Notebook. L'ajout d'un saut de ligne rend l'output plus lisible sur la page, sans que cela ait d'impact sur les informations. Dans ce livre, l'output est aussi présenté sur deux ou plusieurs lignes dans d'autres exemples, même s'il apparaît sur une seule ligne dans Jupyter Notebook.

Les valeurs diffèrent selon le nombre de connexions de chaque sommet. Le sommet 1 ayant quatre connexions (et le plus haut degré), il a aussi la plus forte centralité. Pour plus de clarté, tracez le graphe en appelant la fonction `nx.draw(AGraph, with_labels=True)`, comme le montre la [Figure 8-5](#).

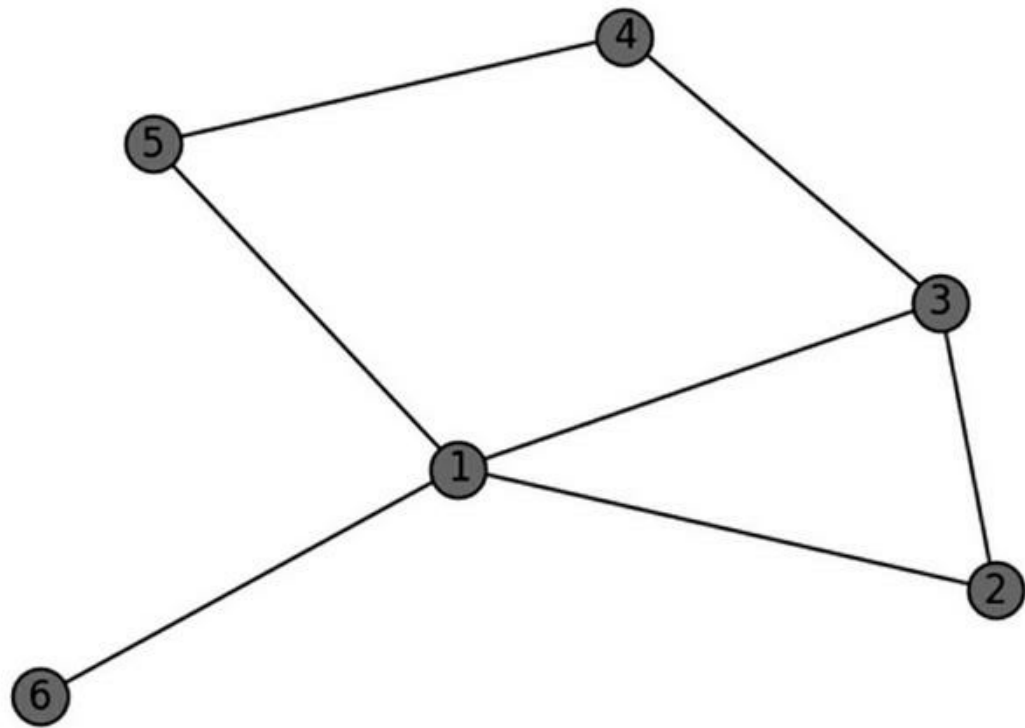


FIGURE 8-5 Tracer le graphe permet de mieux mesurer la centralité de degré.

Le sommet 1 est véritablement au centre du graphe, avec le plus grand nombre de connexions. Compte tenu de son degré, qui est fonction du nombre de connexions, il est le sommet le plus important. Quand vous travaillez sur des graphes orientés, vous pouvez aussi utiliser les fonctions `in_degree centrality()` et `out_degree centrality()` pour déterminer la centralité en fonction du type de connexion plutôt que simplement en fonction du nombre de connexions.

Si vous analysez le trafic automobile, vous devrez peut-être déterminer les points qui sont centraux en fonction de leur distance par rapport à d'autres sommets. Même si un centre commercial en banlieue peut être doté de nombreuses connexions, le fait qu'il soit situé en banlieue risque de réduire son impact sur le trafic. Un supermarché dans le centre-ville avec un nombre réduit de connexions pourra exercer un plus fort impact sur la circulation, sachant qu'il se trouve à proximité d'un grand nombre d'autres sommets. Pour voir comment tout cela fonctionne, ajoutez un autre sommet, le sommet 7, isolé du reste du graphe. Sa centralité sera infinie car il n'est possible de l'atteindre à partir d'aucun sommet. Le code suivant montre comment calculer la centralité liée à la proximité des autres sommets dans notre exemple :

```
AGraph.add_node(7)
nx.closeness centrality(AGraph)

{1: 0.6944444444444445,
2: 0.5208333333333334,
3: 0.5952380952380952,
4: 0.462962962962963,
5: 0.5208333333333334,
6: 0.4166666666666667,
7: 0.0}
```

L'output indique la centralité de chaque sommet du graphe en fonction de sa proximité par rapport aux autres sommets. Le sommet 7 se voit attribuer la valeur 0, ce qui correspond à une distance infinie par rapport à tous les autres sommets. Le

sommet 1, au contraire, se voit attribuer la plus forte valeur, car il est proche de chacun des sommets auxquels il est relié. En calculant la centralité de proximité, on peut déterminer l'importance relative des sommets en fonction de leur localisation.

Une autre forme de centralité liée à la distance est la centralité d'intermédiarité. Supposons que votre entreprise livre de la marchandise au sein d'une agglomération. Il faut que vous sachiez quels sommets sont les plus déterminants pour vos trajets. Vous allez peut-être faire transiter une grande partie des livraisons par un certain sommet. Le calcul de la centralité d'intermédiarité vous permet de déterminer le sommet présentant le nombre le plus élevé de chemins courts entrants. Le code utilisé pour effectuer ce calcul (le sommet 7 isolé étant toujours là) est le suivant :

```
nx.betweenness centrality(AGraph)
```

```
{1: 0.36666666666666664,  
2: 0.0,  
3: 0.13333333333333333,  
4: 0.03333333333333333,  
5: 0.06666666666666667,  
6: 0.0,  
7: 0.0}
```

Comme on pouvait s'y attendre, le sommet 7 est sans effet sur les trajets entre les autres sommets, puisqu'il ne leur est pas relié. De même, le sommet 6 étant un nœud feuille, avec une seule connexion à un autre sommet, n'a aucun effet sur les trajets. Examinons à nouveau la [Figure 8-5](#). Le sous-graphe constitué des sommets 1, 3, 4 et 5 est ici celui qui exerce l'impact le plus déterminant sur les transferts de marchandise. Il n'existe aucune connexion entre les sommets 1 et 4, par conséquent les sommets 3 et 5 servent d'intermédiaires. En l'occurrence, le sommet 2 se comporte comme un nœud feuille.



NetworkX vous fournit un certain nombre d'autres fonctions de calcul de centralité. Vous en trouverez la liste complète à l'adresse <http://networkx.readthedocs.io/en/stable/reference/algorithms.centrality.html>. Il importe de déterminer de quelle manière il est souhaitable de

calculer l'importance. Il est essentiel d'étudier la centralité à la lumière du type d'importance que l'on veut attacher aux sommets et aux arêtes du graphe.

Mettre un graphe sous forme numérique

La précision joue un rôle important en algorithmique. De notre point de vue, trop de précision peut empêcher d'avoir une bonne vue d'ensemble, mais les détails comptent beaucoup pour les ordinateurs. Souvent, plus on leur fournit de détails et meilleurs sont les résultats obtenus. Néanmoins, la forme que prennent ces détails est importante. Pour l'utilisation d'un algorithme, les données doivent être fournies sous un certain format, faute de quoi les résultats obtenus ne seront pas exploitables (ils seront erronés, ou d'autres problèmes pourront se poser).

Heureusement, NetworkX propose un certain nombre de fonctions pour convertir votre graphe sous un format utilisable par d'autres modules ou systèmes. Ces fonctions sont présentées sur la page <http://networkx.readthedocs.io/en/stable/reference/convert.html>.

Les sections suivantes vous montrent comment présenter les données d'un graphe sous la forme d'une matrice NumPy (<http://www.numpy.org/>), d'une représentation éparses SciPy (<https://www.scipy.org/>) ou d'une liste standard Python. Vous utiliserez ces représentations à mesure de votre progression dans la lecture de ce livre pour travailler avec les différents algorithmes (dans les sections suivantes, le code est visible dans le fichier A4D ; 08 ; Graph Conversion.ipynb et il est lié au graphe que vous avez élaboré dans la section « Compter les arêtes et les sommets » de ce chapitre).

Ajouter un graphe à une matrice

À l'aide de NetworkX, vous pouvez facilement transformer votre graphe en matrice NumPy et inversement, selon les exigences des différentes tâches à exécuter. NumPy sert à exécuter toutes sortes de manipulations de données. En analysant les données sur un graphe, vous pouvez constater des tendances qui, autrement, ne seraient pas visibles. Voici le code utilisé pour convertir le graphe en matrice exploitable par NumPy :

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

AGraph = nx.Graph()

Nodes = range(1,6)
Edges = [(1,2), (2,3), (3,4), (4,5), (1,3), (1,5),
(1,6)]

AGraph.add_nodes_from(Nodes)
AGraph.add_edges_from(Edges)

nx.to_numpy_matrix(AGraph)

matrix([[ 0.,  1.,  1.,  0.,  1.,  1.],
        [ 1.,  0.,  1.,  0.,  0.,  0.],
        [ 1.,  1.,  0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.,  1.,  0.],
        [ 1.,  0.,  0.,  1.,  0.,  0.],
        [ 1.,  0.,  0.,  0.,  0.,  0.]])
```

Les lignes et les colonnes obtenues montrent où se trouvent les connexions. Ainsi, par exemple, il n'y a pas de connexion entre le sommet 1 et lui-même, dont on trouve un 0 à l'intersection de la ligne 1 et de la colonne 1. En revanche, il existe une connexion entre le sommet 1 et le sommet 2, par conséquent on trouve un 1 à l'intersection de la ligne 1 et de la colonne 2, ainsi qu'à l'intersection de la ligne 2 et de la colonne 1 (la connexion existant dans les deux sens car elle n'est pas orientée).

La taille de cette matrice dépend du nombre de sommets (le nombre de lignes ou de colonnes est égal au nombre de sommets) et une matrice de grande dimension signifie qu'il y a beaucoup de

sommets à représenter : le nombre total d'éléments de la matrice est le carré du nombre de sommets. Il ne serait pas possible de représenter l'Internet sous cette forme, par exemple, sachant que selon une estimation prudente, il doit exister 10^{10} sites Web et la matrice qui représenterait sa structure devrait contenir 10^{20} éléments, ce qui est au-delà de la capacité actuelle des ordinateurs.

Par ailleurs, le nombre de sommets affecte le contenu de la matrice. Si n est le nombre de sommets, le nombre de chiffres 1 sera au minimum $(n-1)$ et au maximum $n(n-1)$. Selon le nombre de chiffres 1, le graphe sera dense ou clairsemé, ce qui est une chose importante car lorsqu'il y a peu de connexions entre les sommets, comme dans le cas des sites Web, il existe de meilleures solutions pour stocker les données du graphe.

Utiliser les représentations éparses

Le module SciPy sert à différents travaux mathématiques, scientifiques et d'ingénierie. Les données sont mises sous forme de matrice creuse. Une *matrice creuse* est une matrice dans laquelle n'apparaissent que les connexions réelles, toutes les autres entrées étant nulles. L'utilisation d'une matrice creuse, peu gourmande en mémoire, permet d'économiser les ressources. Le code permettant de créer une matrice creuse SciPy à partir d'un graphe NetworkX est le suivant :

```
print(nx.to_scipy_sparse_matrix(AGraph))

(0, 1) 1
(0, 2) 1
(0, 4) 1
(0, 5) 1
(1, 0) 1
(1, 2) 1
(2, 0) 1
(2, 1) 1
(2, 3) 1
(3, 2) 1
(3, 4) 1
```

```
(4, 0) 1
(4, 3) 1
(5, 0) 1
```

Comme on peut le voir, les entrées représentent les coordonnées des arêtes. À chaque coordonnée active est associé un 1. Les coordonnées ont pour base 0. Cela signifie que (0, 1) fait référence à une connexion entre les sommets 1 et 2.

Utiliser une liste pour représenter un graphe

Il se peut que vous ayez aussi besoin de pouvoir créer un dictionnaire de listes. De nombreux développeurs utilisent cette méthode pour écrire un programme qui exécute différentes tâches d'analyse de graphes. Vous pouvez voir un exemple sur la page <https://www.python.org/doc/essays/graphs/>. Le code suivant crée un dictionnaire de listes pour le graphe de notre exemple :

```
nx.to_dict_of_lists(AGraph)

{1: [2, 3, 5, 6], 2: [1, 3], 3: [1, 2, 4], 4: [3, 5],
 5: [1, 4], 6: [1]}
```

Il convient de remarquer que chaque sommet représente une entrée de dictionnaire et est suivi de la liste des sommets auxquels il est relié. Le sommet 1, par exemple, est relié aux sommets 2, 3, 5 et 6.

Chapitre 9

Relier les points

DANS CE CHAPITRE

- » Utiliser des graphes
 - » Exécuter des tâches de tri
 - » Réduire la taille d'un arbre
 - » Trouver le plus court chemin entre deux points
-

Ce chapitre traite de l'utilisation des graphes. Nous utilisons des graphes tous les jours pour réaliser des tâches diverses. Un *graphe* est simplement un ensemble de sommets ou de points reliés par des arêtes, des arcs ou des lignes. Pour parler plus concrètement, chaque carte, chaque plan que vous utilisez est un graphe. Le point de départ, les points intermédiaires et la destination sont des sommets. Ces sommets sont reliés les uns aux autres par les voies de circulation, qui sont des lignes. L'utilisation de graphes vous permet de décrire différentes sortes de relations. Le fonctionnement du GPS est fondé sur la description mathématique des relations entre des points sur la carte et des voies qui assurent les connexions. Une fois que vous aurez fini de lire ce chapitre, vous aurez assimilé les bases à partir desquelles le GPS a été créé (mais peut-être pas les mécanismes de sa concrétisation). Naturellement, une condition fondamentale pour qu'un graphe serve à développer un système GPS est la possibilité de chercher les connexions entre les points sur la carte. C'est le thème de la première section de ce chapitre.

Pour qu'un graphe soit exploitable, il faut l'organiser en ordonnant les sommets, comme expliqué dans la deuxième section de ce chapitre. Autrement, aucune décision ne serait possible. L'algorithme risquerait de boucler ou de produire un résultat non pertinent. Ainsi, par exemple, les premiers GPS ne déterminaient pas toujours correctement la plus courte distance entre deux points, et il arrivait qu'ils ne vous conduisent pas au bon endroit. Ces problèmes étaient liés en partie à la nécessité de trier les données afin qu'elles soient visualisées de la même manière chaque fois que l'algorithme traverse les sommets (pour vous proposer un chemin entre votre domicile et votre lieu de travail).

Quand vous consultez une carte, vous négligez généralement les informations situées dans le coin inférieur droit car votre attention se porte sur les lieux et les itinéraires affichés dans le coin supérieur gauche. L'ordinateur ignore qu'il doit considérer un endroit particulier, tant que vous ne lui avez pas demandé de le faire. Pour que l'attention se porte sur un endroit précis, il faut réduire la taille du graphe, comme l'explique la troisième section de ce chapitre.

Une fois que le problème est ainsi simplifié, l'algorithme peut trouver le plus court chemin entre deux points, comme le décrit la quatrième section de ce chapitre. Il s'agit de ne pas passer plus de temps que nécessaire à circuler entre le domicile et le lieu de travail (dans un sens et dans l'autre). La notion de détermination du plus court chemin étant un peu moins évidente qu'on pourrait le penser, la quatrième section étudie en détail les conditions particulières de la détermination des itinéraires.

Parcourir un graphe de manière efficiente

Parcourir un graphe signifie identifier (et franchir) les sommets (ou vertex) dans un ordre particulier. Ce processus peut consister à la fois à lire les informations et à les mettre à jour. Quand on

parcourt un graphe, les sommets non visités sont des sommets non découverts. Une fois franchis, ils sont découverts, ou traités (l'algorithme ayant traité toutes les arêtes qui en partent). L'ordre de la recherche détermine le type de recherche effectuée, et de nombreux algorithmes peuvent être utilisés pour cette tâche. Les sections qui suivent en présentent deux.

Créer le graphe

Pour comprendre le parcours du graphe, il faut que celui-ci soit tracé. Les exemples de cette section concernent un graphe quelconque, sachant qu'ils doivent avant tout vous donner un aperçu de ces techniques. Le code suivant représente la liste d'adjacence qui se trouve à la fin du [Chapitre 8](#) (vous retrouverez ce code dans le fichier téléchargeable A4D ; 09 ; Graph Traversing.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
graph = {'A': ['B', 'C'],
        'B': ['A', 'C', 'D'],
        'C': ['A', 'B', 'D', 'E'],
        'D': ['B', 'C', 'E', 'F'],
        'E': ['C', 'D', 'F'],
        'F': ['D', 'E']}
```

Ce graphe présente un flux bidirectionnel dont le parcours est A, B, D, F d'un côté (à partir de la racine) et A, C, E, F de l'autre côté (là encore, à partir de la racine). Il y a aussi des connexions (pouvant servir de raccourcis) de B à C, de C à D et de D à E. Avec le module `networkx` présenté au [Chapitre 8](#), vous pouvez faire apparaître graphiquement l'adjacence et la disposition des sommets et des arêtes ([voir Figure 9-1](#)) en utilisant le code suivant :

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
```



```
Graph = nx.Graph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge in graph[node]:
        Graph.add_edge(node, edge)

pos = { 'A': [0.00, 0.50], 'B': [0.25, 0.75],
        'C': [0.25, 0.25], 'D': [0.75, 0.75],
```

ENVISAGER DES REDONDANCES

Quand on parcourt un arbre, chaque chemin se termine par un nœud feuille. Arrivé à ce sommet, on sait que l'on a fini de parcourir le chemin. Cependant, les connexions sont telles qu'il faut parfois franchir certains sommets plus d'une fois pour pouvoir explorer le graphe en totalité. Plus le graphe est dense et plus les possibilités de franchir un même sommet plus d'une fois augmentent. La densité des graphes peut accroître considérablement les besoins de calcul et de stockage.

Afin de limiter les effets négatifs du franchissement répété des mêmes sommets, il est courant de marquer d'une manière ou d'une autre chaque sommet visité, afin qu'à chaque passage d'un sommet on sache si l'algorithme l'a déjà visité. Quand l'algorithme détecte cette situation, il peut simplement sauter le sommet et passer au sommet suivant sur le chemin. Marquer les sommets visités réduit les pertes de performance inhérentes à ce type de redondance.

Marquer les sommets visités permet aussi de vérifier que la recherche est terminée. Autrement, l'algorithme risquerait de continuer indéfiniment à parcourir le graphe.

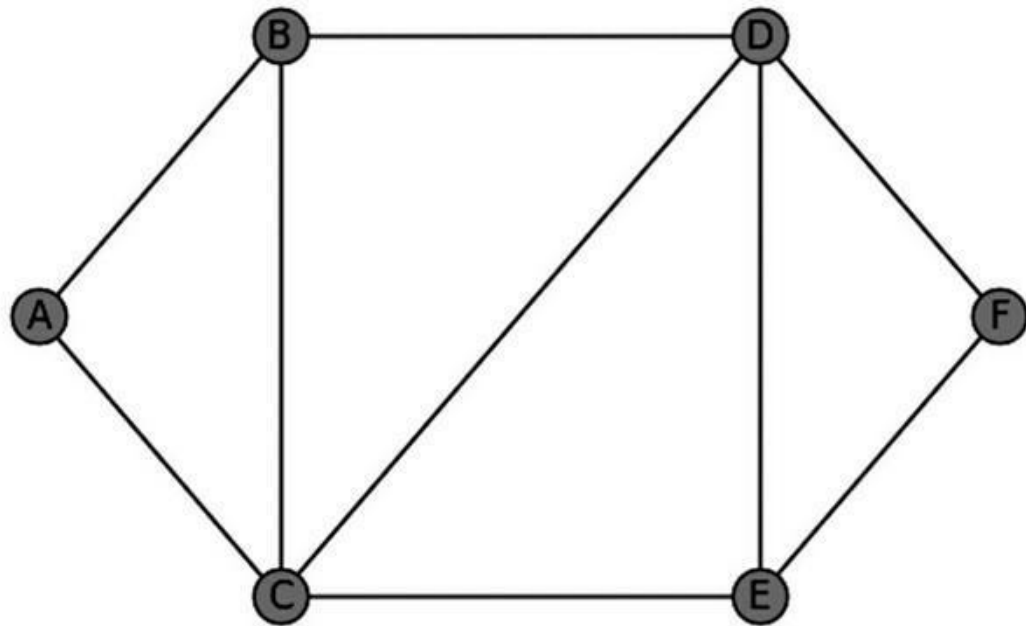


FIGURE 9-1 Représentation de l'exemple par NetworkX.

```
'E': [0.75, 0.25], 'F': [1.00, 0.50]}\n\nnx.draw(Graph, pos, with_labels=True)\nnx.draw_networkx(Graph, pos)\nplt.show()
```

Effectuer un parcours en largeur

Un algorithme de parcours en largeur (BFS) part de la racine du graphe et explore chacun des sommets qui lui sont reliés. Il poursuit sa recherche au niveau suivant, et ainsi de suite jusqu'à la fin du graphe. Dans notre exemple, la recherche commence au point A et se poursuit avec B et C avant d'explorer D. L'algorithme de parcours en largeur parcourt le graphe de façon systématique. La recherche autour d'un sommet se fait de façon circulaire, en parcourant tous les sommets qui lui sont directement reliés, puis ceux qui lui sont reliés par l'intermédiaire d'un sommet, puis par l'intermédiaire de deux sommets, *etc.* Le code correspond à un algorithme de parcours en largeur :

```
def bfs(graph, start):
    queue = [start]
    queued = list()
    path = list()
    while queue:
        print ('La file est : %s' % queue)
        vertex = queue.pop(0)
        print ('Traite %s' % vertex)
        for candidate in graph[vertex]:
            if candidate not in queued:
                queued.append(candidate)
                queue.append(candidate)
                path.append(vertex+'>'+candidate)
                print ('Ajoute %s à la file'
                    % candidate)
    return path

steps = bfs(graph, 'A')
print ('\nBFS:', steps)
```

```
La file est : ['A']
Traite A
Ajoute B à la file
Ajoute C à la file
La file est : ['B', 'C']
Traite B
Ajoute A à la file
Ajoute D à la file
La file est : ['C', 'A', 'D']
Traite C
Ajoute E à la file
La file est : ['A', 'D', 'E']
Traite A
La file est : ['D', 'E']
Traite D
Ajoute F a la file
La file est : ['E', 'F']
Traite E
La file est : ['F']
Traite F
```

```
BFS: ['A>B', 'A>C', 'B>A', 'B>D', 'C>E', 'D>F']
```



L'output montre comment l'algorithme effectue la recherche. Elle se fait dans l'ordre prévu, un niveau à la fois. Le principal avantage du BFS est la garantie d'obtenir le plus court chemin

entre deux points.



Dans notre exemple, une liste simple est utilisée comme file. Comme cela est expliqué au [Chapitre 4](#), une file est une structure de données de type premier entré/ premier sorti (FIFO) dont la logique est la même que celle d'une file d'attente devant un guichet de banque : le premier élément entré dans la file et aussi le premier à en sortir. À cet effet, Python propose une structure de données plus adaptée encore, le deque (on prononce « dek »), que l'on crée à l'aide de la fonction deque qui se trouve dans le module collections. Elle effectue des insertions et des extractions en temps linéaire, et s'utilise à la fois comme file et comme pile. Pour en savoir plus sur la fonction deque, consultez la page <https://pymotw.com/2/collections/deque.html>.

Opter pour le parcours en profondeur

Outre le parcours en largeur (BFS), vous pouvez utiliser le parcours en profondeur (DFS) pour découvrir les sommets d'un graphe. L'algorithme d'un DFS commence la recherche à la racine du graphe et explore chaque sommet en suivant un chemin unique jusqu'au point d'arrivée. Ensuite, il revient en arrière pour explorer les autres chemins jusqu'à ce qu'il ait atteint la racine. Si d'autres chemins sont disponibles à partir de la racine, l'algorithme en sélectionne un et entreprend à nouveau la recherche. Le principe est d'explorer chaque chemin en totalité avant d'en explorer un autre. Pour que cette technique de recherche soit viable, il faut que l'algorithme marque chaque sommet visité. Ainsi, il sait quels sommets doivent encore être visités et il peut déterminer le prochain chemin à suivre. Le parcours en largeur (BFS) et le parcours en profondeur (DFS) peuvent donner un résultat différent, selon la manière dont le graphe doit être traversé. Du point de vue de la programmation, la différence entre ces deux algorithmes est dans la manière dont chacun stocke les sommets pour effectuer le parcours suivant :

- » Une file dans le cas du BFS, c'est-à-dire une liste fonctionnant selon le principe FIFO. Les sommets nouvellement découverts sont rapidement traités.
- » Une pile dans le cas du DFS, c'est-à-dire une liste fonctionnant selon le principe LIFO (dernier entré/premier sorti).

Le code suivant crée un DFS :

```
def dfs(graph, start):
    stack = [start]
    parents = {start: start}
    path = list()
    while stack:
        print ('La pile est : %s' % stack)
        vertex = stack.pop(-1)
        print ('Traite %s' % vertex)
        for candidate in graph[vertex]:
            if candidate not in parents:
                parents[candidate] = vertex
                stack.append(candidate)
                print ('Ajoute %s à la pile'
                    % candidate)
        path.append(parents[vertex]+'>'+vertex)
    return path[1:]

steps = dfs(graph, 'A')
print ('\nDFS:', steps)
```

```
La pile est : ['A']
Traite A
Ajoute B à la pile
Ajoute C à la pile
La pile est : ['B', 'C']
Traite C
Ajoute D à la pile
Ajoute E à la pile
La pile est : ['B', 'D', 'E']
Traite E
Ajoute F à la pile
La pile est : ['B', 'D', 'F']
Traite F
La pile est : ['B', 'D']
Traite D
```

```
La pile est : ['B']  
Traite B
```

```
DFS: ['A>C', 'C>E', 'E>F', 'C>D', 'A>B']
```

La première ligne d'output montre l'ordre réel de la recherche. La recherche commence à la racine, comme prévu, mais elle suit le côté gauche du graphe. La dernière étape consiste à parcourir la seule branche extérieure à la boucle, en l'occurrence le sommet D.

Il convient de noter que l'output n'est pas le même qu'avec le BFS. Ici, le traitement commence par le sommet A et suit le côté opposé du graphe, vers le sommet F. Le programme revient ensuite vers la racine pour parcourir les autres chemins possibles. Comme cela a été évoqué, le cheminement diffère lorsqu'une pile est utilisée plutôt qu'une file. Avec une pile, ce type de recherche peut aussi utiliser la récursivité. L'algorithme donne alors des résultats plus rapidement qu'avec un BFS. L'inconvénient est que l'on utilise davantage de mémoire.



Quand l'algorithme utilise une pile, il exploite le dernier résultat disponible (dans l'alternative, il exploiterait le premier résultat placé dans la file). Les fonctions récursives produisent un résultat puis s'appellent elles-mêmes en utilisant ce résultat. Une pile fait exactement la même chose dans une itération : l'algorithme produit un résultat, ce résultat est placé au sommet d'une pile, puis il est immédiatement retiré de la pile pour être traité.

Savoir quelle application utiliser

Le choix entre BFS et DFS dépend de la manière dont vous comptez exploiter le résultat de la recherche. Les développeurs utilisent souvent le BFS pour déterminer le plus rapidement possible le plus court chemin entre deux points. Le BFS est donc communément utilisé dans des applications comme le GPS, lorsque le plus court chemin est primordial. Dans le cadre de ce livre, le BFS sera aussi utilisé pour l'arbre couvrant, le plus court chemin et divers autres algorithmes de minimisation.

Le DFS consiste à déterminer un chemin complet avant d'en explorer un autre. On l'utilise pour une recherche détaillée, plutôt que générale. C'est pourquoi cette technique est souvent utilisée dans les jeux, où il est important de trouver un chemin complet. C'est aussi la méthode optimale lorsqu'il s'agit, par exemple, de trouver la solution d'un labyrinthe.



Le choix entre BFS et DFS doit parfois se faire en fonction des limites de chacune des deux techniques. Le BFS nécessite une grande quantité de mémoire car il stocke systématiquement tous les chemins avant de trouver une solution. Le DFS utilise moins de mémoire, mais n'offre pas la garantie d'obtenir le chemin le plus court et le plus direct.

Trier les éléments du graphe

L'efficacité de la recherche d'un chemin sur un graphe dépend de l'ordre des données. Imaginons que dans une bibliothèque, les livres soient rangés dans un ordre fantaisiste : trouver un ouvrage donné demanderait des heures de recherche. Au contraire, le rangement des livres selon un classement logique est fondamental.

Une autre propriété des bibliothèques est importante également quand on utilise certains types de graphes. Quand vous recherchez un livre, vous vous intéressez à une catégorie spécifique, puis vous sélectionnez un rayonnage, puis une étagère, *etc.* La recherche se fait depuis le moins spécifique vers le plus spécifique, et vous ne parcourez pas à nouveau les niveaux précédents. Vous ne risquez donc pas de chercher dans des rayonnages qui n'ont rien à voir avec le sujet qui vous intéresse.

Les sections suivantes sont consacrées aux *graphes acycliques orientés*, qui sont des graphes finis et orientés ne comportant aucune boucle. En d'autres termes, le parcours commence en un certain point et se poursuit sur un chemin donné jusqu'à une destination, sans jamais revenir au point de départ. Avec un tri topologique, le parcours se fait toujours des sommets précédents

vers les sommets suivants. Ce type de graphe trouve toutes sortes d'applications pratiques, comme par exemple les horaires.

DES GRAPHES AVEC DES BOUCLES

Dans l'exécution d'un processus, il peut être nécessaire que certaines étapes soient répétées. Quand vous lavez votre voiture, par exemple, vous rincez la carrosserie, vous l'aspergez de savon, puis vous rincez à nouveau. Vous allez ensuite trouver un endroit encore sale : il faut alors savonner à nouveau, puis rincer, et vérifier que la saleté est partie. Peut-être allez-vous répéter le processus une nouvelle fois, et peut-être encore, jusqu'à ce que cette partie de la carrosserie soit enfin propre. C'est le principe de la boucle : créer une situation dans laquelle une série d'étapes se répète :

- » Soit jusqu'à ce qu'une condition particulière soit vérifiée : la saleté est partie.
- » Soit jusqu'à ce que le processus ait été répété un nombre de fois déterminé.

Travailler avec des graphes acycliques orientés

Compte tenu de leurs nombreuses applications pratiques, les graphes acycliques orientés font partie des types de graphes les plus importants. Ils obéissent aux principes fondamentaux suivants :

- » Leur parcours suit un ordre particulier, si bien qu'une fois parcouru un chemin d'un sommet vers un autre, il n'est pas possible de revenir au sommet précédent par quelque chemin que ce soit.
- » Il existe un chemin particulier d'un sommet à un autre, ce qui permet de définir une série de chemins prédictible.

Les graphes acycliques orientés sont couramment utilisés dans les contextes organisationnels. Un arbre généalogique, par exemple, est un graphe acyclique orienté. Même lorsqu'une activité ne se déroule pas de façon primordiale selon un ordre chronologique ou autre, ce type de graphe permet de définir des chemins prédictibles. C'est pourquoi ces graphes sont particulièrement faciles à traiter.

Dans les graphes acycliques orientés, certaines étapes peuvent cependant être facultatives. Imaginons que vous prépariez des sandwiches. La première étape est de disposer d'une tranche de pain. Vous pouvez la tartiner et ajouter des condiments, ou bien vous pouvez y placer directement la tranche d'emmental. Au bout du compte, vous obtiendrez toujours un sandwich, mais les chemins vers le point d'arrivée sont variés. Après avoir ajouté la tranche d'emmental, vous pouvez éventuellement ajouter une feuille de salade, par exemple, avant de compléter le sandwich par la seconde tranche de pain. Vous suivez un chemin particulier, mais il existe plusieurs possibilités de passage d'une étape à la suivante.



Jusqu'à présent, nous avons étudié plusieurs configurations de graphes, et certaines peuvent se combiner : un graphe peut être orienté, pondéré et dense :

- » **Orienté** : Les arêtes sont à sens unique, et un graphe orienté peut aussi être :
 - **Cyclique** : Les arêtes forment une boucle qui ramène au sommet initial après la visite des sommets intermédiaires.
 - **Acyclique** : Le graphe ne comporte aucune boucle.
- » **Non orienté** : Les arêtes relient les sommets dans les deux sens.
- » **Pondéré** : À chaque arête est associé un coût, qui peut s'exprimer en temps, en argent ou en énergie.

- » **Non pondéré** : Il n'y a aucun coût, ou toutes les arêtes présentent le même coût.
- » **Dense** : Se dit d'un graphe qui comporte un grand nombre d'arêtes par rapport au nombre de sommets.
- » **Creux** : se dit d'un graphe qui comporte un nombre réduit d'arêtes par rapport au nombre de sommets.

L'utilité du tri topologique

Un aspect remarquable des graphes acycliques orientés est la variété considérable d'activités qu'ils permettent de représenter. Cependant, dans certaines activités, les tâches doivent être envisagées dans un ordre spécifique. C'est là qu'intervient le tri topologique, qui consiste à ordonner tous les sommets du graphe avec des arcs orientés de gauche à droite. Ainsi, le programme peut facilement effectuer la traversée du graphe et traiter les sommets l'un après l'autre, dans le bon ordre.

Avec le tri topologique, le graphe est organisé de telle sorte que chaque sommet conduise à un sommet suivant. Si l'on élabore le calendrier de la construction d'un bâtiment, par exemple, on ne commencera pas par le toit pour redescendre jusqu'aux fondations. C'est par les fondations qu'il faut commencer, et chaque étage représentera une étape. Une fois le deuxième étage terminé, on construira le troisième étage, et ensuite on ne reprendra pas le deuxième. Du troisième, on passera au quatrième, et ainsi de suite. Dans toute programmation de ce type, le chemin s'effectue entre un certain point de départ et un certain point d'arrivée, grâce à un graphe acyclique orienté assorti d'un tri topologique.

Le tri topologique vous permet de vérifier l'absence de boucle dans un graphe (en présence d'une boucle, il ne serait pas possible d'ordonner les arcs reliant les sommets de gauche à droite, sachant qu'au moins un des sommets serait relié à un sommet précédent). Le tri topologique est utile aussi pour les algorithmes qui traitent

des graphes complexes, sachant qu'il indique le meilleur ordre pour leur traitement.

On peut obtenir un tri topologique en utilisant l'algorithme de parcours DFS. Il faut simplement noter l'ordre dans lequel il traite les sommets. Dans l'exemple précédent, l'output apparaît dans l'ordre suivant : A, C, E, F, D, B. En suivant la séquence de la [Figure 9-1](#), vous remarquerez que le tri topologique suit les arcs du périmètre extérieur du graphe et effectue un tour complet : après avoir atteint le dernier sommet du tri topologique, on se trouve à une étape de A, qui est le point de départ de la séquence.

Réduire à un arbre couvrant minimum

Les problèmes qui sont résolus par des algorithmes consistent souvent à définir le minimum de ressources à utiliser, par exemple lorsque l'on recherche un moyen économique d'atteindre tous les points sur une carte. Ce type de problème revêtait une importance particulière à la fin du XIX^e siècle et au début du XX^e siècle, au moment où les réseaux électriques et ferroviaires commençaient à apparaître dans un certain nombre de pays et à révolutionner les transports et les modes de vie. Le recours aux compagnies privées était onéreux (il fallait beaucoup de temps et de main-d'œuvre). Pour utiliser le moins de matériel possible, consommer le moins d'heures de main-d'œuvre et réaliser ainsi des économies, il fallait éviter les connexions redondantes.



Certaines redondances sont souhaitables sur les infrastructures essentielles de transport et d'énergie, même si l'on recherche les solutions les plus économiques. En effet, si un réseau ne pouvait être connecté que d'une seule façon, il pourrait facilement se retrouver hors service, soit accidentellement, soit à la suite d'un acte volontaire (par exemple un acte de guerre), et un grand nombre de clients ne seraient plus desservis.

En 1926, le mathématicien tchèque Otakar Borůvka avait trouvé une solution pour développer un réseau électrique en utilisant la moins grande quantité possible de câblage. Sa solution était très efficace, car non seulement elle permettait de trouver comment relier les villes de Moravie (aujourd'hui la partie orientale de la République tchèque) de la façon la plus économique possible, mais elle présentait une complexité en temps de $O(m * \log n)$, m étant le nombre d'arcs (ici, les câbles électriques) et n le nombre de sommets (ici, les villes). Depuis, d'autres chercheurs ont amélioré la solution de Borůvka (en fait, les spécialistes de l'algorithmique l'avaient partiellement oubliée et l'ont ensuite redécouverte). Même si les algorithmes que vous pouvez trouver dans les livres sont mieux conçus et plus faciles à étudier (ceux de Prim et de Kruskal), ils ne donnent pas de meilleurs résultats en termes de complexité en temps.

Le problème consistant à trouver le moyen le plus économique d'accomplir une tâche donnée est représenté par un arbre couvrant minimal. Un *arbre couvrant minimal* est la liste des arêtes nécessaires pour relier tous les sommets dans un graphe non orienté. Un même graphe peut contenir plusieurs arbres couvrants, selon sa configuration, et déterminer combien d'arbres il contient peut devenir un problème complexe. Dans un graphe, chaque chemin parcouru du point de départ au point d'arrivée est un arbre couvrant. Un arbre couvrant visite chaque sommet une fois seulement : il ne boucle pas et aucun élément de chemin n'est répété.

Dans un graphe non orienté, les arbres couvrants ont la même longueur. Dans les graphes non pondérés, toutes les arêtes ont la même longueur, et l'ordre du parcours n'a pas d'importance car le chemin est toujours le même. Tous les arbres couvrants possibles ont le même nombre d'arêtes, $n-1$ (n étant le nombre de sommets), lesquelles ont exactement la même longueur. Par ailleurs, tout algorithme de parcours d'un graphe, de type BFS ou DFS, suffit pour trouver tous les arbres couvrants possibles.

Les choses se compliquent quand on travaille sur un graphe pondéré, dont les arêtes ont des longueurs différentes. Dans ce

cas, parmi les différents arbres couvrants possibles, très peu, ou même un seul, présente la longueur minimum possible. Un *arbre couvrant minimal* est un arbre couvrant qui garantit un chemin dans lequel la somme des poids des arêtes est la plus réduite possible. Un graphe non orienté ne comporte généralement qu'un seul chemin couvrant minimal, mais là encore, tout dépend de sa configuration. Pour bien vous représenter ce qu'est un arbre couvrant minimal, songez que sur une carte, vous pouvez trouver un certain nombre de chemins possibles pour aller du point A au point B. Sur chaque itinéraire, il y a des intersections où vous devez tourner ou changer de route, et ce sont autant de sommets. La distance entre deux sommets est le poids de l'arête qui les relie. En général, l'arête qui relie un point A à un point B représente le plus court chemin entre ces deux points.

Cependant, un arbre couvrant minimal ne correspond pas nécessairement à la solution évidente. Quand vous consultez une carte, par exemple, la distance n'est pas toujours pour vous le critère essentiel : il se peut que vous préféreriez prendre en compte le temps de trajet, ou le coût en carburant et en péage, ou d'autres éléments encore, et à chacun de ces critères peut correspondre un arbre couvrant minimal complètement différent. Dans ce contexte, les sections qui suivent vous permettent de mieux appréhender la notion d'arbre couvrant minimal et vous indiquent comment trouver le poids minimal dans un problème de graphe. Pour la détermination d'un arbre couvrant minimal avec Python, l'ajout de poids sur les arêtes du graphe précédent se fait à l'aide du code suivant (vous le retrouverez dans le fichier téléchargeable A4D ; 09 ; Minimum Spanning Tree.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

graph = {'A': {'B':2, 'C':3},
        'B': {'A':2, 'C':2, 'D':2},
        'C': {'A':3, 'B':2, 'D':3, 'E':2},
        'D': {'B':2, 'C':3, 'E':1, 'F':3},
```

```

'E': {'C':2, 'D':1, 'F':1},
'F': {'D':3, 'E':1}}

Graph = nx.Graph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge, weight in graph[node].items():
        Graph.add_edge(node,edge, weight=weight)

pos = { 'A': [0.00, 0.50], 'B': [0.25, 0.75],
        'C': [0.25, 0.25], 'D': [0.75, 0.75],
        'E': [0.75, 0.25], 'F': [1.00, 0.50]}

labels = nx.get_edge_attributes(Graph,'poids')
nx.draw(Graph, pos, with_labels=True)
nx.draw_networkx_edge_labels(Graph, pos,
                             edge_labels=labels)
nx.draw_networkx(Graph,pos)
plt.show()

```

La [Figure 9-2](#) représente le graphe avec une valeur attribuée à chaque arête. Cette valeur peut représenter par exemple une quantité de temps, ou de carburant, ou un coût monétaire. Les graphes pondérés représentent les situations dans lesquelles on peut rejoindre un sommet ou le quitter. C'est pourquoi ils peuvent représenter tous les problèmes d'optimisation possibles dans un espace géographique (comme par exemple un problème d'itinéraire entre des villes).

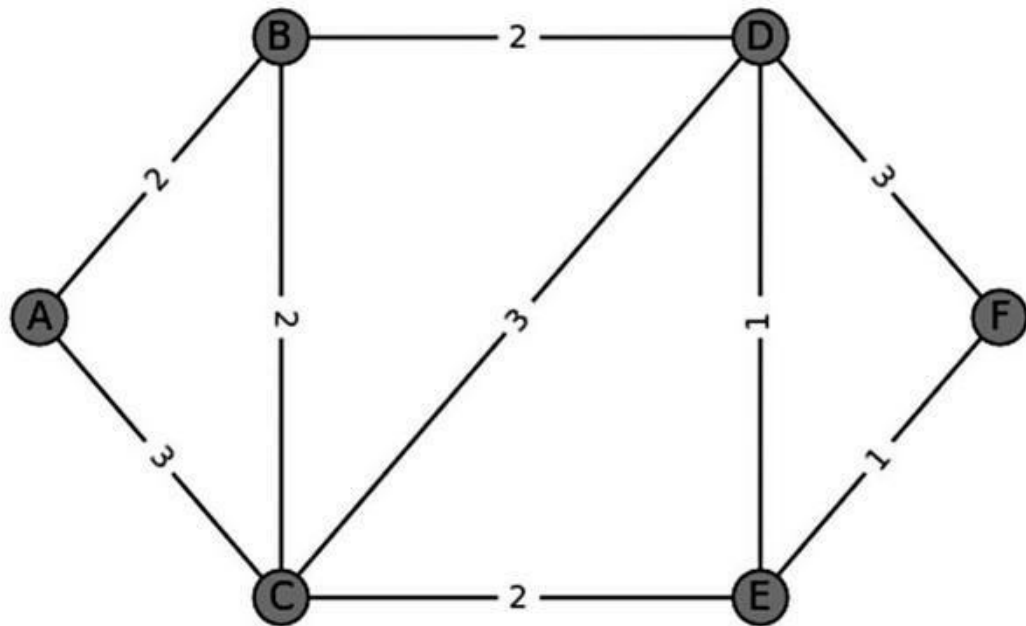


FIGURE 9-2 Quand le graphe de notre exemple devient un graphe pondéré

Dans cet exemple, il est intéressant de remarquer que toutes les arêtes sont affectées de coefficients positifs. Or, les graphes pondérés peuvent comporter des arêtes affectées de poids négatifs. C'est un avantage dans un certain nombre de situations, par exemple quand un mouvement entre deux sommets peut engendrer soit un gain, soit une perte, ou quand un processus chimique peut entraîner une libération d'énergie.



Tous les algorithmes ne sont pas adaptés à la gestion de coefficients négatifs. Il est important de noter que certains ne peuvent fonctionner qu'avec des coefficients positifs.

Savoir quels sont les bons algorithmes à utiliser

Différents algorithmes peuvent servir à créer un arbre couvrant minimal. Les plus courants sont les algorithmes gloutons, qui fonctionnent en temps polynomial. Le temps polynomial est une

puissance du nombre d'arêtes, par exemple $O(n^2)$ ou $O(n^3)$ (pour plus de détails sur le temps polynomial, voir la Cinquième partie). Les principaux facteurs dont dépend la vitesse d'exécution de ces algorithmes sont liés au processus de décision : le fait qu'une arête appartienne à l'arbre couvrant minimal, ou le fait que le poids total minimal de l'arbre résultant dépasse une certaine valeur. Dans ce contexte, voici des exemples d'algorithmes pour la résolution d'un problème d'arbre couvrant minimal :

- » **L'algorithme de Borůvka** : Inventé par Otakar Borůvka en 1926 pour résoudre le problème consistant à trouver le moyen optimal de distribuer l'électricité en Moravie. Cet algorithme parcourt un graphe et identifie à chaque étape les arêtes affectées des poids les moins élevés. Les calculs commencent au niveau de chaque sommet pour trouver, parmi les arêtes qui y conduisent, celle de plus faible poids, après quoi les chemins sont combinés pour former ce que l'on appelle des forêts, constituées d'arbres élémentaires, de manière à déterminer un chemin combinant toutes les forêts avec un poids minimal.
- » **L'algorithme de Prim** : Initialement inventé par Jarník en 1930, il a été redécouvert par Prim en 1957. Cet algorithme prend comme point de départ un sommet choisi arbitrairement et construit l'arbre couvrant minimal, arête par arête, en sélectionnant toujours celle de poids minimal.
- » **L'algorithme de Kruskal** : Développé par Joseph Kruskal en 1956, il tient à la fois de l'algorithme de Borůvka (création de forêts constituées d'arbres) et de celui de Prim (recherche de l'arête de poids minimal pour chaque sommet, et création de forêts arête par arête).
- » **L'algorithme de parcours inverse et d'élimination** : Il s'agit en fait de l'algorithme de Kruskal à l'envers. Il n'est pas souvent utilisé.



Ces algorithmes sont des algorithmes gloutons, un type d'algorithme évoqué au [Chapitre 2](#) et présenté en détail au

[Chapitre 15](#). L'*algorithme glouton* trouve progressivement une solution en prenant, à chaque étape, la meilleure décision possible, sans y revenir. Ainsi, par exemple, pour déterminer le plus court chemin passant par un certain nombre de sommets, l'algorithme glouton sélectionne systématiquement le plus court chemin entre deux sommets qui se suivent.

Utiliser des files de priorité

Plus loin dans ce chapitre, nous allons voir comment mettre en œuvre les algorithmes de Prim et de Kruskal pour obtenir un arbre couvrant minimal, et l'algorithme de Dijkstra pour déterminer le plus court chemin dans un graphe en utilisant Python. Cependant, il faut au préalable disposer d'une méthode pour trouver, parmi un ensemble d'arêtes, celles qui présentent un poids minimal. Cette opération implique un tri, or trier des éléments demande du temps. Elle n'est pas simple, comme nous l'avons vu au [Chapitre 7](#). Dans ces exemples, les arêtes font l'objet d'un tri répété, aussi est-il pratique de disposer d'une structure de données appelée la *file de priorité*.

Le principe des *files de priorité* est celui des structures de données arborescentes par tas permettant un tri rapide des éléments insérés. Comme le chapeau magique du magicien, les tas prioritaires stockent les arêtes avec leurs poids respectifs et fournissent immédiatement l'arête de poids minimal parmi les éléments stockés.

Cet exemple utilise une classe permettant des comparaisons entre files prioritaires pour déterminer si la file contient des éléments et si, parmi ces éléments, il se trouve une certaine arête (pour éviter les doubles insertions). La file prioritaire présente une autre caractéristique utile (dont l'utilité est expliquée dans le cadre de l'utilisation de l'algorithme de Dijkstra) : quand est insérée une arête dont le poids est différent de ce qui est déjà stocké, le programme met à jour le poids de l'arête et réorganise sa position dans le tas.

```

from heapq import heapify, heappop, heappush

class priority_queue():
    def __init__(self):
        self.queue = list()
        heapify(self.queue)
        self.index = dict()
    def push(self, priority, label):
        if label in self.index:
            self.queue = [(w,l)
                           for w,l in self.queue if l!=label]
            heapify(self.queue)
            heappush(self.queue, (priority, label))
        self.index[label] = priority
    def pop(self):
        if self.queue:
            return heappop(self.queue)
    def __contains__(self, label):
        return label in self.index
    def __len__(self):
        return len(self.queue)

```

Exploiter l'algorithme de Prim



L'algorithme de Prim produit l'arbre couvrant minimal dans un graphe grâce à un parcours sommet par sommet. En partant d'un sommet quelconque, il ajoute les arêtes en utilisant une contrainte, que si le sommet fait partie de l'arbre couvrant et si le sommet suivant n'en fait pas partie, le poids de l'arête qui les relie doit être le plus faible possible. En procédant de cette manière, il est impossible de créer des cycles dans l'arbre couvrant (il faudrait pour cela ajouter une arête reliant deux sommets qui feraient déjà partie tous les deux de l'arbre couvrant) et l'on a la garantie d'obtenir un arbre minimal puisqu'on ajoute les arêtes dont le poids est le moins élevé. En termes d'étapes, l'algorithme inclut les trois phases suivantes, la dernière étant itérative :

1. **Enregistrer les arêtes d'un arbre couvrant minimal et les sommets utilisés lorsqu'ils sont inclus dans la solution.**

2. Partir d'un sommet quelconque du graphe et l'inclure dans la solution.

3. Déterminer s'il existe encore des sommets qui ne sont pas inclus dans la solution :

- Énumérer les arêtes qui touchent les sommets inclus dans la solution.
- Insérer l'arête de poids minimal dans l'arbre couvrant (il s'agit là du principe de l'algorithme glouton : toujours choisir le minimum à chaque étape afin de minimiser le résultat global).

Vous pouvez tester l'algorithme pour le graphe pondéré de notre exemple en traduisant ces étapes sous forme du code Python suivant :

```
def prim(graph, start):
    treepath = {}
    total = 0
    queue = priority_queue()
    queue.push(0 , (start, start))
    while queue:
        weight, (node_start, node_end) = queue.pop()
        if node_end not in treepath:
            treepath[node_end] = node_start
            if weight:
                print("Ajout arête de %s" \
                    " à %s de poids %i"
                    % (node_start, node_end, weight))
            total += weight
        for next_node, weight \
            in graph[node_end].items():
            queue.push(weight , (node_end, next_node))
    print ("Longueur totale de l'arbre couvrant : %i" %
        total)
    return treepath
```

```
treepath = prim(graph, 'A')
```

```
Ajout arête de A à B de poids 2
Ajout arête de B à C de poids 2
Ajout arête de B à D de poids 2
Ajout arête de D à E de poids 1
```

Ajout arête de E à F de poids 1
Longueur totale de l'arbre couvrant : 8

Le programme affiche les étapes du traitement, en montrant à chaque étape quelle arête il ajoute et quel poids cette arête ajoute au poids total. Dans cet exemple, la somme totale des poids est affichée et l'algorithme fournit un dictionnaire Python contenant le sommet final comme clé et le sommet de départ comme valeur pour chaque arête de l'arbre couvrant résultant. Une autre fonction, `represent_tree`, transforme en tuples les paires clé/valeur du dictionnaire, puis trie les tuples en vue d'une meilleure lisibilité du chemin :

```
def represent_tree(treepath):  
    progression = list()  
    for node in treepath:  
        if node != treepath[node]:  
            progression.append((treepath[node], node))  
    return sorted(progression, key=lambda x:x[0])  
  
print (represent_tree(treepath))  
  
[('A', 'B'), ('B', 'C'), ('B', 'D'), ('D', 'E'),  
 ('E', 'F')]
```



La fonction `represent_tree` réorganise l'output de l'algorithme de Prim en vue d'une meilleure lisibilité. Cependant, l'algorithme est associé à un graphe non orienté, c'est-à-dire sur lequel les arêtes peuvent être parcourues dans les deux sens. Cette hypothèse est prise en compte sachant qu'il n'y a aucun contrôle du sens des arêtes à ajouter à la file de priorité pour un traitement ultérieur.

Tester l'algorithme de Kruskal

L'algorithme de Kruskal utilise la stratégie des algorithmes gloutons, tout comme celui de Prim, mais il sélectionne les arêtes de coefficient minimal parmi l'ensemble des arêtes (alors que l'algorithme de Prim compare les arêtes en fonction des sommets de l'arbre couvrant). Pour déterminer si une arête doit faire partie

de la solution, l'algorithme exécute un processus d'agrégation des sommets. Quand une arête est liée à un sommet déjà inclus dans la solution, l'algorithme la rejette afin d'éviter de créer une boucle. L'algorithme procède de la façon suivante :

- 1. Réunir toutes les arêtes dans un tas et les trier de telle sorte que les plus courtes soient sur le dessus.**
- 2. Créer une série d'arbres constitués chacun d'un seul sommet (ainsi, le nombre d'arbres est égal au nombre de sommets). Les arbres sont reliés sous forme d'un agrégat jusqu'à ce qu'ils fusionnent pour former un arbre unique de longueur minimale couvrant l'ensemble des sommets.**
- 3. Répéter les opérations suivantes jusqu'à ce que la solution ne contienne plus autant d'arêtes qu'il y a de sommets dans le graphe :**
 - a. Choisir l'arête la plus courte du tas.
 - b. Déterminer si les deux sommets reliés par cette arête apparaissent dans des arbres différents, parmi l'ensemble des arbres reliés.
 - c. Lorsque les arbres sont différents, les relier à l'aide de l'arête (ce qui définit une agrégation).
 - d. Lorsque les sommets apparaissent dans le même arbre, rejeter l'arête.
 - e. Répéter les étapes a à d pour les arêtes qui restent dans le tas.

L'exemple suivant montre comment traduire ces étapes sous forme de code dans Python :

```
def kruskal(graph):  
    priority = priority_queue()  
    print ("Pousse toutes les arêtes dans la file de  
priorité")  
    treepath = list()  
    connected = dict()
```

```

for node in graph:
    connected[node] = [node]
    for dest, weight in graph[node].items():
        priority.push(weight, (node, dest))
print ("Au total %i arêtes" % len(priority))
print ("Éléments reliés : %s"
      % connected.values())

total = 0
while len(treepath) < (len(graph)-1):
    (weight, (start, end)) = priority.pop()
    if end not in connected[start]:
        treepath.append((start, end))
        print ("Somme des éléments %s et %s : "
              % (connected[start], connected[end]))
        print ("\tAjout arête de %s " \
              "à %s de poids %i"
              % (start, end, weight))
        total += weight
        connected[start] += connected[end][:]
        for element in connected[end]:
            connected[element] = connected[start]
        print ("Longueur totale de l'arbre couvrant : %i" %
              total)
    return sorted(treepath, key=lambda x:x[0])

print ('\nArbre couvrant minimal : %s' %
      kruskal(graph))

```

```

Pousse toutes les arêtes dans la file de priorité
Au total 9 arêtes
Éléments reliés : dict_values([['A'], ['E'], ['F'],
                               ['B'], ['D'], ['C']])
Somme des éléments ['E'] et ['D'] :
    Ajout arête de E à D de poids 1
Somme des éléments ['E', 'D'] et ['F'] :
    Ajout arête de E à F de poids 1
Somme des éléments ['A'] et ['B'] :
    Ajout arête de A à B de poids 2
Somme des éléments ['A', 'B'] et ['C'] :
    Ajout arête de B à C de poids 2
Somme des éléments ['A', 'B', 'C'] et ['E', 'D', 'F']
:
    Ajout arête de B à D de poids 2
Longueur totale de l'arbre couvrant : 8

Arbre couvrant minimal :
[('A', 'B'), ('B', 'C'), ('B', 'D'), ('E', 'D'),

```

('E', 'F']



L'algorithme de Kruskal offre une solution similaire à celle proposée par l'algorithme de Prim. Cependant, des graphes différents peuvent donner des solutions différentes pour l'arbre couvrant minimal lorsque l'on utilise les algorithmes de Prim et de Kruskal, sachant que ces algorithmes ne procèdent pas de la même manière. Des méthodes différentes impliquent souvent des arbres couvrants minimaux différents.

Déterminer quel algorithme fonctionne le mieux

L'algorithme de Prim et celui de Kruskal produisent un unique composant relié, qui joint tous les sommets du graphe en utilisant les chemins les moins longs (l'arbre couvrant minimal). En additionnant les poids des arêtes, on peut déterminer la longueur de l'arbre couvrant résultant. Les deux algorithmes donnant toujours une solution viable, les critères pour choisir le meilleur sont le temps d'exécution et la capacité de l'un ou de l'autre à traiter un type de graphe pondéré donné.

Concernant le temps d'exécution, les deux algorithmes donnent des résultats similaires avec un taux de complexité de Big-O de $O(E * \log(V))$, où E est le nombre d'arêtes et V le nombre de sommets. Cependant, il faut tenir compte de la manière dont ils résolvent le problème, car il existe des différences en termes de temps d'exécution moyen prévisible.

L'algorithme de Prim construit progressivement une solution unique en ajoutant des arêtes, tandis que l'algorithme de Kruskal crée un ensemble de solutions partielles et les rassemble. L'algorithme de Prim s'appuie sur des structures de données plus complexes, car il ajoute continuellement des arêtes au stock dans lequel il les sélectionne, et choisit toujours l'arête la plus courte pour progresser vers sa solution. Sur un graphe dense, l'algorithme de Prim est préférable à celui de Kruskal car sa file

de priorité, basée sur les tas, effectue les tâches de tri de façon rapide et efficace.



Notre exemple utilise une file de priorité basée sur un tas binaire pour exécuter la lourde tâche de sélection des arêtes les plus courtes, mais il existe des structures de données encore plus performantes comme le *tas de Fibonacci*, qui peut produire des résultats plus rapides quand le tas contient un grand nombre d'arêtes. En utilisant le tas de Fibonacci, la complexité d'exécution de l'algorithme de Prim peut devenir de type $O(E + V * \log(V))$, ce qui est évidemment avantageux lorsqu'il y a beaucoup d'arêtes (la composante E est alors additionnée et non pas multipliée), par rapport au temps d'exécution $O(E * \log(V))$ noté précédemment.

L'algorithme de Kruskal n'a pas vraiment besoin d'une file de priorité (même si un des exemples en utilise une), car l'énumération et le tri des arêtes n'ont lieu qu'une seule fois, au début du processus. Basé sur des structures de données plus simples qui sont appliquées aux arêtes triées, il constitue le choix idéal pour les graphes ordinaires et les graphes creux, ceux qui comptent un nombre limité d'arêtes.

Trouver le plus court chemin

Le plus court chemin entre deux points n'est pas nécessairement la ligne droite, surtout s'il n'y a pas de ligne droite dans votre graphe. Supposons que vous deviez installer un réseau de lignes électriques dans une collectivité locale. La logique du plus court chemin voudrait que les lignes soient rectilignes d'un point à un autre, sans considération pour ce qui peut se trouver sur leur trajectoire. Dans le monde réel, une solution aussi simple est généralement inenvisageable. Les câbles doivent être enfouis sous les voies de communication et ne doivent pas traverser des propriétés privées. Il s'agit donc de trouver des chemins de manière à réduire le plus possible les distances.

Préciser ce que signifie trouver le plus court chemin

Pour trouver le plus court chemin, il existe de nombreuses applications. Il s'agit de déterminer le chemin correspondant à la plus courte distance entre un point A et un point B. La détermination du plus court chemin est utile dans le domaine des transports (comment arriver à destination en consommant le moins de carburant) comme dans le domaine des communications (comment faire transiter l'information pour qu'elle arrive le plus rapidement possible). De façon moins évidente, le problème du plus court chemin trouve aussi des applications dans le traitement d'image (pour couper des contours), les jeux (comment parvenir au but en jouant le plus petit nombre de coups) et bien d'autres domaines dans lesquels le problème peut être modélisé par un graphe pondéré, orienté ou non.

L'algorithme de Dijkstra, qui permet de résoudre le problème du plus court chemin, a trouvé de nombreuses applications. Edsger W. Dijkstra, un informaticien néerlandais, avait conçu cet algorithme en 1959 à titre de démonstration de la puissance de traitement d'un nouvel ordinateur appelé ARMAC (<http://www-set.win.tue.nl/UnsungHeroes/machines/armac.html>). À l'origine, l'algorithme trouvait la plus petite distance entre 64 villes des Pays-Bas en se basant sur un graphe simple.



Il existe d'autres algorithmes permettant de résoudre le problème du plus court chemin. Celui de Bellman-Ford et celui de Floyd-Warshall sont plus complexes, mais ils peuvent traiter des graphes comportant des poids négatifs (les poids négatifs permettent de mieux représenter certains problèmes). Ces deux algorithmes sortent du cadre de ce livre, mais vous trouverez plus de détails sur le site <https://www.hackerearth.com/ja/practice/algorithms/graphs/shortest-path-algorithms/tutorial/>. Sachant que le problème du plus court chemin fait intervenir des graphes qui sont à la fois pondérés et orientés, le graphe de notre exemple nécessite une nouvelle

adaptation avant d’aller plus loin (la [Figure 9-3](#) montre le résultat). Vous retrouverez ce code dans le fichier téléchargeable A4D ; 09 ; Shortest Path.ipynb sur le site *Dummies* (pour plus de détails, consultez l’Introduction).

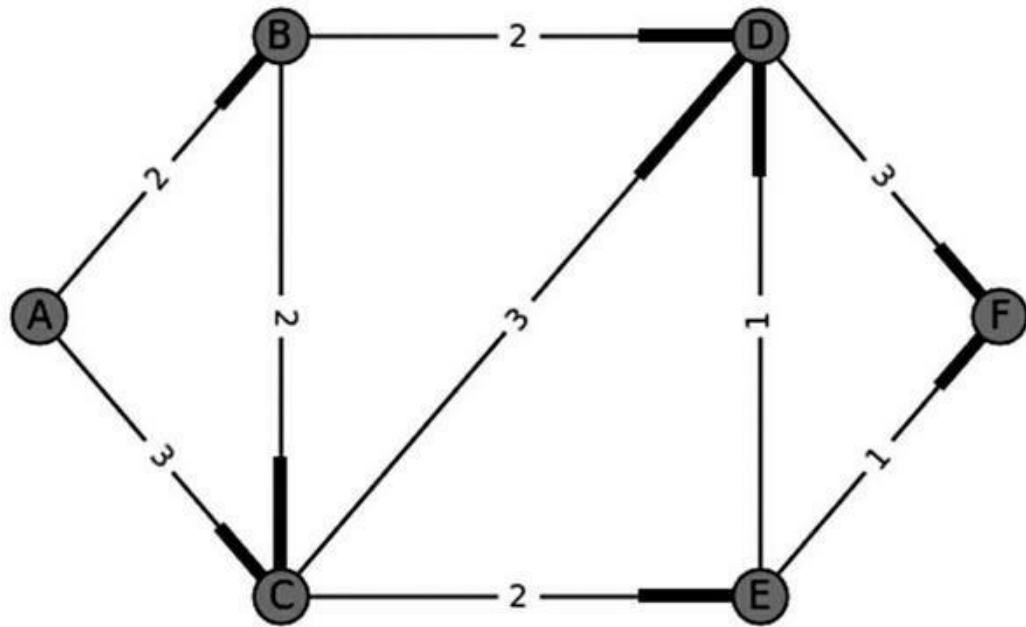


FIGURE 9-3 Le graphe de notre exemple est maintenant pondéré et orienté.

```

import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

graph = {'A': {'B':2, 'C':3},
        'B': {'C':2, 'D':2},
        'C': {'D':3, 'E':2},
        'D': {'F':3},
        'E': {'D':1, 'F':1},
        'F': {}}

Graph = nx.DiGraph()
for node in graph:
    Graph.add_nodes_from(node)
    for edge, weight in graph[node].items():
        Graph.add_edge(node,edge, weight=weight)

```

```
pos = { 'A': [0.00, 0.50], 'B': [0.25, 0.75],
        'C': [0.25, 0.25], 'D': [0.75, 0.75],
        'E': [0.75, 0.25], 'F': [1.00, 0.50]}

labels = nx.get_edge_attributes(Graph, 'weight')
nx.draw(Graph, pos, with_labels=True)
nx.draw_networkx_edge_labels(Graph, pos,
                             edge_labels=labels)

nx.draw_networkx(Graph, pos)
plt.show()
```

Expliquer l'algorithme de Dijkstra

L'algorithme de Dijkstra nécessite en input un sommet de départ et (éventuellement) un sommet d'arrivée. Si vous ne lui donnez pas le sommet d'arrivée, l'algorithme calcule la plus courte distance entre le sommet de départ et n'importe quel autre sommet du graphe. Si vous précisez le sommet d'arrivée, le processus d'exploration du graphe s'arrête à la lecture de ce sommet et l'algorithme donne le résultat à ce point, quelle que soit la partie du graphe qui reste inexplorée.

L'algorithme commence par estimer la distance entre le point de départ et les autres sommets. Il s'agit du paramètre de base qu'il enregistre dans la file de priorité et qui, par convention, est réglé sur l'infini. L'algorithme procède ensuite à l'exploration des sommets voisins, de façon similaire à un algorithme BFS. Cela lui permet de déterminer quels sommets sont à proximité. La distance entre deux sommets est le poids de l'arc qui les relie. Le programme stocke cette information dans la file de priorité en mettant à jour les poids de façon appropriée.



Naturellement, si l'algorithme peut explorer des sommets voisins, c'est parce qu'un arc orienté les relie au sommet de départ. L'algorithme de Dijkstra tient compte du sens des arcs.

À ce stade, l'algorithme passe au sommet le plus proche dans le graphe, déterminé en fonction de l'arc le plus court dans la file de priorité. Techniquement, l'algorithme *visite* un nouveau sommet.

Il commence par explorer les sommets voisins en excluant ceux qu'il a déjà visités, détermine le coût de la visite de chaque sommet non visité, et compare la distance jusqu'à ces sommets à la distance enregistrée dans la file de priorité.

Quand la distance dans la file de priorité est infinie, cela signifie qu'il s'agit de la première visite du sommet en question et que l'algorithme va maintenant enregistrer la distance la plus courte. Quand la distance enregistrée dans la file de priorité n'est pas infinie, mais est supérieure à la distance que l'algorithme vient de calculer, cela signifie que l'algorithme a trouvé un raccourci, un chemin plus court pour atteindre ce sommet depuis le point de départ, et qu'il stocke cette information dans la file de priorité. Naturellement, si la distance enregistrée dans la file de priorité est plus courte que celle qu'il vient de calculer, l'algorithme ne stocke pas cette nouvelle donnée, sachant que le nouveau chemin est plus long. Après avoir mis à jour toutes les distances vers les sommets voisins, l'algorithme détermine s'il a atteint le sommet d'arrivée. Si ce n'est pas le cas, il sélectionne l'arc le plus court parmi ceux qui se trouvent dans la file de priorité, il le visite, puis il calcule la distance vers chacun des nouveaux sommets voisins.



Comme cela a été expliqué, l'algorithme de Dijkstra tient une comptabilité précise du coût du trajet vers chaque sommet qu'il trouve, et il ne met à jour ses données que lorsqu'il trouve un chemin plus court. La complexité d'exécution de l'algorithme dans la notation Big-O est $O(E * \log(V))$, où E est le nombre d'arcs et V le nombre de sommets dans le graphe. Le code suivant montre comment implémenter l'algorithme de Dijkstra en utilisant Python :

```
def dijkstra(graph, start, end):
    inf = float('inf')
    known = set()
    priority = priority_queue()
    path = {start: start}

    for vertex in graph:
        if vertex == start:
            priority.push(0, vertex)
```

```

        else:
            priority.push(inf, vertex)
last = start

while last != end:
    (weight, actual_node) = priority.pop()
    if actual_node not in known:
        for next_node in graph[actual_node]:
            upto_actual = priority.index[actual_node]
            upto_next = priority.index[next_node]
            to_next = upto_actual + \
graph[actual_node][next_node]
            if to_next < upto_next:
                priority.push(to_next, next_node)
                print("Trouvé raccourci de %s à %s"
                    % (actual_node, next_node))
                print ("\Longueur totale à ce point : %i"
                    % to_next)
                path[next_node] = actual_node

            last = actual_node
            known.add(actual_node)

    return priority.index, path
dist, path = dijkstra(graph, 'A', 'F')

Trouvé raccourci de A à C
    Longueur totale à ce point : 3
Trouvé raccourci de A à B
    Longueur totale à ce point : 2
Trouvé raccourci de B à D
    Longueur totale à ce point : 4
Trouvé raccourci de C à E
    Longueur totale à ce point : 5
Trouvé raccourci de D à F
    Longueur totale à ce point : 7
Trouvé raccourci de E à F
    Longueur totale à ce point : 6

```

L'algorithme retourne quelques données utiles : le plus court chemin jusqu'à la destination et les distances minimales enregistrées pour les sommets visités. Pour visualiser le chemin le plus court, il vous faut la fonction `reverse_path`, qui réorganise le chemin pour le rendre lisible :

```
def reverse_path(path, start, end):
```

```
    progression = [end]
    while progression[-1] != start:
        progression.append(path[progression[-1]])
    return progression[: :-1]
print (reverse_path(path, 'A', 'F'))

['A', 'C', 'E', 'F']
```

Vous pouvez aussi connaître la distance la plus courte jusqu'à chaque sommet rencontré, en interrogeant le dictionnaire dist :

```
print (dist)

{'D': 4, 'A': 0, 'B': 2, 'F': 6, 'C': 3, 'E': 5}
```

Chapitre 10

Découvrir les secrets des graphes

DANS CE CHAPITRE

- » Envisager les réseaux sociaux sous forme de graphes
 - » Interagir avec le contenu d'un graphe
-

Le [Chapitre 8](#) vous apporte les bases de la théorie des graphes appliquée aux mathématiques. Le [Chapitre 9](#) vous permet d'approfondir vos connaissances en développant les relations entre les graphes et les algorithmes. Le présent chapitre porte sur l'application des théories de ces deux chapitres précédents sous forme d'utilisation pratique des graphes.

La première section est une approche des réseaux sociaux par le biais des graphes. Il est important d'étudier les connexions qui découlent de ces réseaux. L'analyse des conversations, par exemple, peut révéler des tendances et vous permettre de comprendre le thème sous-jacent mieux que par une simple lecture de ces conversations. Il se peut qu'un sujet attire l'attention davantage qu'un autre, en raison de sa plus grande importance. Naturellement, il est nécessaire de procéder à ce type d'analyse quand on traite des problèmes comme le spam. L'analyse peut aboutir à toutes sortes de conclusions intéressantes. Elle peut permettre, par exemple, d'orienter les dépenses de publicité de manière à attirer l'attention le plus possible et ainsi, mieux vendre.

La seconde section étudie le parcours des graphes en vue de résultats spécifiques. Au volant, par exemple, vous avez besoin de savoir quel est le meilleur itinéraire entre deux points sachant que le chemin le plus court ne sera pas toujours le plus rapide, notamment s'il y a des travaux sur la chaussée. Il faut parfois randomiser la recherche pour trouver le meilleur itinéraire ou la meilleure conclusion, un problème qui est également abordé dans cette section.

Envisager les réseaux sociaux comme des graphes

Toute interaction sociale est nécessairement liée à toutes les interactions sociales du même type. Prenons l'exemple d'un réseau social comme Facebook. Les liens qui se trouvent sur votre page vous relient à des amis et à des proches, mais aussi à des sources extérieures qui, elles-mêmes, sont reliées à d'autres sources extérieures. Chacun de vos amis et proches entretient lui aussi des liens extérieurs. Toutes ces connexions directes et indirectes entre différentes pages font que toutes les pages sont reliées les unes aux autres, même si le processus consistant à passer d'une page à une autre peut faire appel à une myriade de liens. La connectivité prend toutes sortes de formes. Ce qu'il faut retenir ici, c'est qu'il est difficile d'étudier les réseaux sociaux simplement en visualisant des pages Facebook ou d'autres sources d'informations. L'*analyse des réseaux sociaux* consiste à étudier les interactions qui ont lieu sur ces réseaux à l'aide de graphes appelés *sociogrammes*, dans lesquels les sommets (qui peuvent être les pages Facebook) sont représentés par des points et les liaisons (notamment les liens vers des pages externes) par des lignes. Les sections qui suivent sont consacrées à des questions liées à l'étude des réseaux sociaux par le biais des graphes.

Les regroupements sur les réseaux

sociaux

Les utilisateurs forment des communautés : ils se regroupent en fonction de leurs idées et de leur vision du monde. En étudiant ces regroupements, il devient plus facile d'attribuer certains comportements à un groupe dans son ensemble (tandis qu'attribuer un comportement à un individu est à la fois dangereux et douteux). L'étude des regroupements repose sur l'idée que les personnes entre lesquelles des connexions se sont créées partagent souvent des idées et des aspirations. L'appartenance à un groupe sert d'indicateur. Ainsi, par exemple, on observe souvent des regroupements autour de thèmes comme la détection de la fraude à l'assurance ou les impôts. Lorsque des regroupements sont inattendus, ils peuvent donner lieu à des suspicions : en l'absence des raisons habituelles pour lesquelles les gens forment un réseau, on peut se demander s'il ne s'agit pas de personnes malhonnêtes.

Les *graphes de l'amitié* représentent la manière dont les utilisateurs de l'Internet nouent des liens. Les sommets représentent les individus et les arêtes représentent leurs liens qui peuvent être familiaux, amicaux ou professionnels. Ce sont généralement des graphes non orientés, puisqu'ils représentent des relations mutuelles, et ils peuvent être pondérés lorsqu'ils représentent l'intensité des liens entre deux personnes.



De nombreuses études portent sur des graphes non orientés, qui concernent uniquement les associations entre individus. On peut aussi utiliser des graphes orientés pour représenter le fait que l'individu A connaisse l'individu B sans que l'individu B connaisse l'existence de l'individu A. Dans ce cas, il y a en réalité 16 différentes sortes de triades à étudier. Par souci de simplification, ce chapitre n'aborde que les quatre types suivants : relations fermées, ouvertes, paire connectée et triade non connectée.

Dans un graphe de l'amitié, les connexions entre les sommets dans ces regroupements dépendent de triades, qui sont

fondamentalement des triangles d'un genre particulier. Les connexions entre trois individus se répartissent entre les catégories suivantes :

- » **Relation fermée** : Les trois individus se connaissent, comme c'est le cas, par exemple, de trois membres d'une même famille.
- » **Relation ouverte** : Un des trois individus connaît les deux autres, mais les deux autres ne se connaissent pas. C'est le cas, par exemple, si vous êtes en relation avec un collègue au travail et avec une amie, et si ce collègue et cette amie ne se connaissent pas.
- » **Paire connectée** : Un des trois individus connaît un des deux autres, mais pas le troisième. Il s'agit en fait de deux individus qui se connaissent et qui font la connaissance d'un troisième, susceptible de vouloir faire partie de leur groupe.
- » **Triade non connectée** : La triade constitue un groupe, mais dans lequel aucun membre ne connaît les autres. Ce dernier cas de figure peut sembler un peu curieux, mais c'est une situation qui se produit à l'occasion, par exemple, d'une convention ou d'un séminaire. Les participants forment un groupe, mais chacun peut tout ignorer des autres. Néanmoins, sachant qu'ils ont des centres d'intérêt similaires, on peut utiliser la technique du regroupement pour étudier le comportement du groupe.

Des triades se forment naturellement dans les relations humaines, et les réseaux sociaux sur Internet exploitent souvent ce principe pour accélérer les connexions entre les participants. La densité des connexions est importante dans tous les cas, sachant que sur un réseau connecté, l'information se diffuse et se partage plus facilement. Ainsi, quand LinkedIn a décidé d'accroître la densité des connexions sur son réseau social professionnel (<https://www.linkedin.com/>), cette société a d'abord recherché les triades ouvertes afin de les fermer en invitant les gens à créer des

liens. La fermeture des triades est au fondement même de l'*algorithme de suggestion de connexions* de LinkedIn. Pour en savoir plus sur son fonctionnement, lisez la réponse de Quora sur la page : <https://www.quora.com/How-does-LinkedIn-People-You-May-Know-work>.

L'exemple présenté dans cette section est inspiré du graphe du réseau social Zachary's Karate Club : voir la page <https://networkdata.ics.uci.edu/data.php?id=105>. Il s'agit d'un petit graphe qui permet de voir comment les réseaux fonctionnent sans devoir consacrer du temps à charger un vaste jeu de données. Heureusement, le jeu de données est inclus dans le module `networkx` présenté au [Chapitre 8](#). Le réseau du club de karaté de Zachary regroupe les relations amicales entre 34 membres d'un club de karaté entre 1970 et 1972. Le sociologue Wayne W. Zachary en a fait le sujet d'une étude et a publié un article intitulé « An Information Flow Model for Conflict and Fission in Small Groups ». Un fait intéressant concernant ce graphe et l'article en question est qu'à cette époque, un conflit était survenu entre un des moniteurs de karaté (sommet numéro 0) et le président du club (sommet numéro 33). En effectuant un regroupement sur le graphe, vous pouvez presque parfaitement prédire la division du club en deux groupes peu de temps après cet incident.

Dans cet exemple, il s'agit de tracer un graphe faisant apparaître les groupes (ainsi vous pouvez les visualiser plus facilement), c'est pourquoi vous devez utiliser aussi le module `matplotlib`. Le code suivant montre comment représenter les sommets et les arêtes pour ce jeu de données (vous retrouverez ce code dans le fichier téléchargeable A4D ; 10 ; Social Networks.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

graph = nx.karate_club_graph()

pos=nx.spring_layout(graph)
nx.draw(graph, pos, with_labels=True)
```

```
plt.show()
```

Pour afficher le graphe à l'écran, vous devez aussi fournir une configuration qui détermine la position des sommets. Cet exemple utilise l'algorithme de dessin basé sur les forces de Fruchterman-Reingold (avec appel de la fonction `nx.spring_layout`). Cependant, vous pouvez choisir une des autres configurations présentées dans la section « Graph Layout » sur la page <https://networkx.github.io/documentation/networkx-1.9/reference/drawing.html>. La [Figure 10-1](#) représente l'output pour cet exemple (votre output peut être légèrement différent).



L'algorithme de dessin basé sur les forces de Fruchterman-Reingold produit automatiquement des graphiques lisibles, sur lesquels les sommets et les arêtes, bien séparés, ne se croisent généralement pas. Ces graphiques sont à l'image de ce qui se produit en physique entre des particules porteuses de charges électriques ou entre des aimants de même signe. Sur le graphe qui apparaît à l'écran, on constate que certains sommets n'ont qu'une connexion, d'autres en ont deux, et d'autres en ont plus de deux. Les arêtes forment des triades, comme mentionné précédemment. Cependant, le plus important est de remarquer, comme le montre clairement la [Figure 10-1](#), le regroupement qui se produit sur un réseau social.

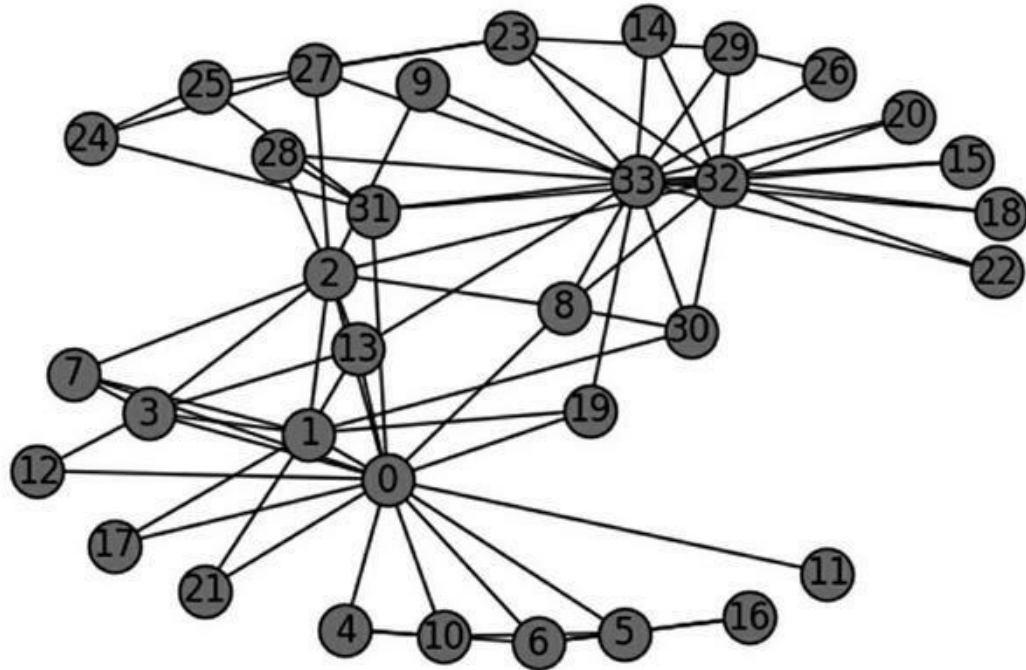


FIGURE 10-1 Un graphe faisant apparaître le regroupement des relations entre amis sur un réseau social.

Où l'on rencontre des communautés

Souvent, un groupe de personnes présentant des liens étroits est assimilable à une communauté. On utilise le terme de *clique* à propos d'un groupe dans lequel l'appartenance est exclusive et les membres se connaissent tous très bien. Pour la plupart d'entre nous, nous avons fait partie d'un groupe d'amis, à l'école ou au lycée par exemple, et nous aimions nous réunir. Nous formions alors une clique.



On peut identifier des cliques sur les graphes non orientés. Les graphes orientés présentent une nette distinction entre les composantes connectées lorsqu'il existe un lien direct entre tous les sommets pairs dans la composante même. La ville est un exemple de composante fortement connectée, car il est possible d'atteindre n'importe quelle destination à partir de n'importe quel point de départ en suivant des voies à sens unique et à double

Dans cet exemple, le programme commence par simplement extraire du jeu de données du club de karaté les sommets ayant au minimum quatre connexions, puis affiche les cliques dont la taille minimum est de quatre. Naturellement, vous pouvez fixer le nombre minimum de connexions que vous voulez, selon ce que vous voulez obtenir. Peut-être considérez-vous comme une clique une communauté dans laquelle chaque élément a vingt connexions, tandis que pour d'autres, trois connexions suffisent.

La liste des cliques ne vous est cependant pas très utile si ce sont les communautés qui vous intéressent. Pour les voir, vous devez utiliser des algorithmes spécialisés et complexes qui fusionneront les cliques qui se recoupent et trouveront des regroupements, par exemple la méthode de la percolation de cliques décrite sur la page <https://gaplogs.net/2012/04/01/simplecommunity-detection-algorithms/>. Le module NetworkX comporte `k_clique_communities`, une application de l'algorithme de percolation de cliques qui donne l'union de toutes les cliques d'une certaine taille (le paramètre `k`). Ces cliques d'une certaine taille partagent `k-1` éléments (elles diffèrent d'un seul élément, ce qui constitue une règle vraiment stricte).

La percolation de cliques produit la liste de toutes les communautés trouvées. Dans notre exemple, une clique s'est formée autour du moniteur de karaté, l'autre autour du président du club. Par ailleurs, vous pouvez extraire tous les sommets qui font partie d'une communauté sous forme d'un ensemble unique, et ainsi, créer un sous-graphe constitué uniquement de communautés.

Enfin, vous pouvez tracer le sous-graphe et l'afficher. La [Figure 10-2](#) représente l'output dans notre exemple, constitué de l'ensemble des cliques comportant au moins quatre connexions.

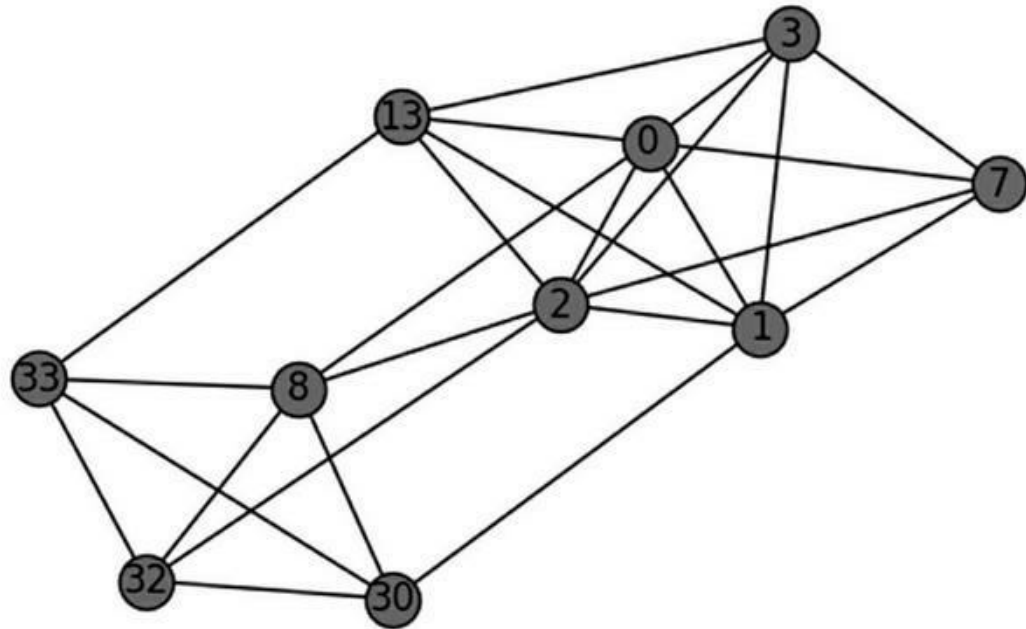


FIGURE 10-2 Les communautés comportent souvent des cliques qui peuvent se révéler utiles pour l'analyse des réseaux sociaux.

Trouver des cliques dans les graphes est un problème complexe qui implique beaucoup de calculs (c'est un problème difficile) qu'un algorithme permet de résoudre par une recherche par force brute, c'est-à-dire en passant en revue tous les sous-ensembles de sommets possibles afin de déterminer si ce sont des cliques.

Avec un peu de chance, sachant qu'une certaine randomisation est nécessaire pour que l'algorithme fonctionne, vous pourrez obtenir une grande clique en utilisant une méthode simple dont le degré de complexité est $O(n+m)$, où n est le nombre de sommets et m le nombre d'arêtes. Les étapes qui suivent décrivent le processus.

1. **Trier les sommets par degré (nombre de connexions) décroissant.**
2. **Placer dans la clique le sommet de plus haut degré (autre possibilité, choisir au hasard un des sommets de plus haut degré).**
3. **Répéter les étapes 1 et 2 jusqu'à ce qu'il n'y ait plus de sommets à examiner.**

4. Vérifier que le prochain sommet fait partie de la clique :

- S'il fait partie de la clique, l'ajouter à la clique.
- S'il ne fait pas partie de la clique, répéter le test sur les sommets qui restent.

À la fin, après plusieurs déroulements de l'algorithme, vous devez obtenir la liste des sommets constituant la plus grande clique présente dans le graphe.

Parcourir un graphe

Parcourir ou *traverser* un graphe signifie visiter chacun de ses sommets. Le parcours d'un graphe peut servir à déterminer le contenu d'un sommet ou à le mettre à jour en fonction des besoins. Au cours de la traversée d'un graphe, il est tout à fait possible de visiter certains sommets plus d'une fois, en raison de la connectivité qui le caractérise. C'est pourquoi il peut être nécessaire de marquer les sommets au fur et à mesure qu'ils sont visités, une fois que l'on a pris connaissance de leur contenu. Le parcours d'un graphe est important pour déterminer la façon dont les sommets sont connectés, en vue d'exécuter diverses tâches. Les chapitres précédents étudient les techniques de base du parcours des graphes. Les sections suivantes expliquent quelques techniques de parcours de graphes parmi les plus avancées.

Compter les degrés de séparation

Dans un graphe, le *degré de séparation* est la distance entre les sommets. Dans un graphe non orienté et non pondéré, chaque arête représente un degré de séparation. Cependant, dans d'autres sortes de graphes comme les cartes, où chaque arête peut représenter une distance ou un laps de temps, les degrés de séparation peuvent être très différents. L'idée générale est que les degrés de séparation indiquent une distance. L'exemple traité dans cette section (de même que celui qui va suivre) utilise les données

de graphe suivantes (vous le retrouverez dans le fichier téléchargeable A4D ; 10 ; Graph Navigation.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction) :

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

data = {'A': ['B', 'F', 'H'],
        'B': ['A', 'C'],
        'C': ['B', 'D'],
        'D': ['C', 'E'],
        'E': ['D', 'F', 'G'],
        'F': ['E', 'A'],
        'G': ['E', 'H'],
        'H': ['G', 'A']}

graph = nx.DiGraph(data)
pos=nx.spring_layout(graph)
nx.draw_networkx_labels(graph, pos)
nx.draw_networkx_nodes(graph, pos)
nx.draw_networkx_edges(graph, pos)
plt.show()
```

Il s'agit d'une version plus développée du graphe utilisé au [Chapitre 6](#). La [Figure 10-3](#) montre comment ce graphe se présente, ce qui vous permet de visualiser l'effet de l'appel de fonction. Il convient de noter que c'est un graphe orienté (networkx DiGraph). En effet, l'utilisation d'un graphe orienté comporte certains avantages lorsqu'il s'agit de déterminer les degrés de séparation (ou d'effectuer toutes sortes d'autres calculs).

Pour déterminer les degrés de séparation entre deux éléments, il faut un point de départ. Pour les besoins de cet exemple, vous pouvez partir du sommet A. Le code suivant est l'appel de fonction du module networkx suivi de son output :

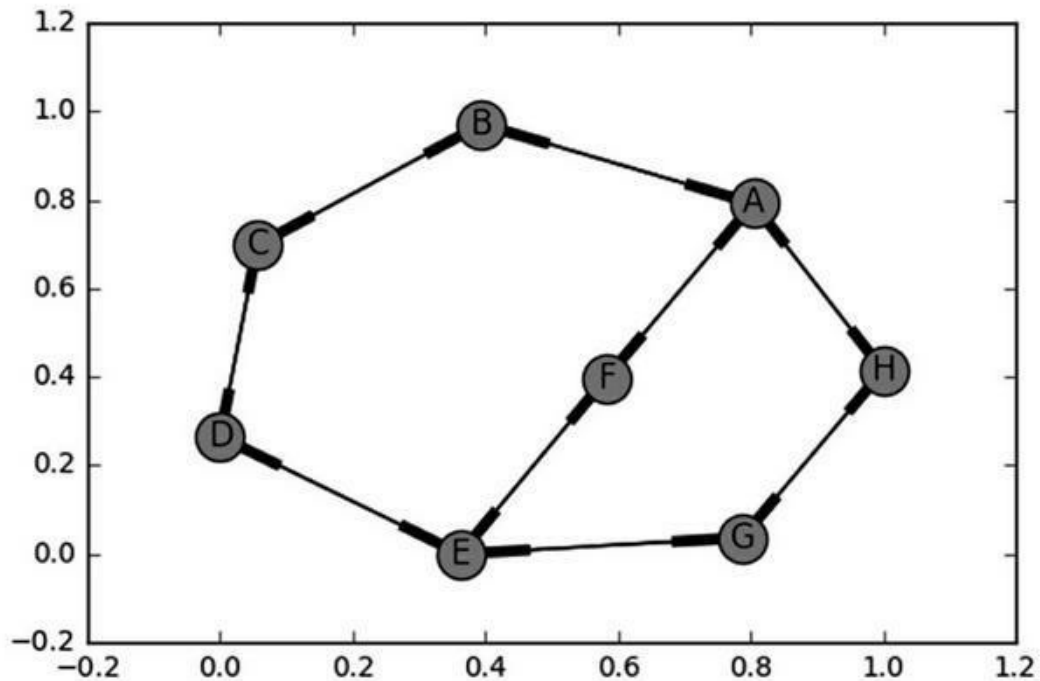


FIGURE 10-3 Un graphe utilisé pour un parcours.

```
nx.shortest_path_length(graph, 'A')
```

```
{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 2, 'F': 1, 'G':  
2,  
'H': 1}
```

La distance entre le sommet A et le sommet A est évidemment 0. Le plus haut degré de séparation est 3, entre le sommet A et le sommet D. Ce genre d'information vous permet de déterminer l'itinéraire à suivre ou de comparer le coût en carburant et le temps de trajet d'un itinéraire à un autre. Il peut en effet être très important de connaître la distance entre deux points. Le module `networkx` utilisé dans cet exemple se retrouve dans un grand nombre d'algorithmes de mesure de distances, comme on peut le voir sur la page

<https://networkx.github.io/documentation/development/reference/a>



Pour mesurer l'importance qu'il y a à utiliser un graphe orienté quand on doit calculer des degrés de séparation, supprimez la connexion entre les sommets A et F. Modifiez les données comme

suit :

```
data = {'A': ['B', 'H'],
        'B': ['A', 'C'],
        'C': ['B', 'D'],
        'D': ['C', 'E'],
        'E': ['D', 'F', 'G'],
        'F': ['E', 'A'],
        'G': ['E', 'H'],
        'H': ['G', 'A']}
```

Cette fois, quand on appelle la fonction `nx.shortest_path_length`, l'output devient très différent, car il n'est plus possible d'aller directement de A à F :

```
{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 3, 'F': 4, 'G':
2,
'H': 1}
```

On remarquera que la disparition de ce chemin a entraîné un changement dans certains degrés de séparation. La distance jusqu'au sommet F est maintenant la plus longue, elle est égale à 4.

Parcourir un graphe de façon aléatoire

Il peut vous arriver de devoir parcourir un graphe de façon aléatoire. Parcourir un graphe de façon aléatoire plutôt que rechercher un itinéraire particulier peut servir à simuler certaines activités naturelles, par exemple le parcours d'un animal en quête de nourriture. Le parcours aléatoire peut également intervenir dans toutes sortes d'autres activités intéressantes, comme les jeux. Cependant, il peut aussi présenter des aspects pratiques. Supposons que vous soyez pris dans un embouteillage en raison d'un accident et qu'en conséquence, l'itinéraire le plus court ne soit plus envisageable. Dans certains cas, le choix d'une alternative au hasard peut être la bonne option.



Le module `networkx` ne vous permet pas d'obtenir directement un chemin aléatoire. Il vous permet cependant de trouver tous les itinéraires disponibles, après quoi vous pouvez en choisir un au hasard dans la liste. Le code suivant en est l'illustration avec le graphe de la section précédente.

```
import random
random.seed(0)

paths = nx.all_simple_paths(graph, 'A', 'H')

path_list = []
for path in paths:
    path_list.append(path)
    print("Itinéraire possible : ", path)

sel_path = random.randint(0, len(path_list) - 1)

print("L'itinéraire choisi est : ",
      path_list[sel_path])

Itinéraire possible : ['A', 'B', 'C', 'D', 'E', 'G', 'H']
Itinéraire possible : ['A', 'H']
Itinéraire possible : ['A', 'F', 'E', 'G', 'H']
L'itinéraire choisi est : ['A', 'H']
```

Le code assigne une valeur particulière à la graine afin que l'on obtienne à chaque fois le même résultat. Cependant, en changeant cette valeur, on peut obtenir des résultats différents de ceux de notre exemple de code. En effet, le graphe simple de la [Figure 10-3](#) offre trois itinéraires pour se rendre du sommet A au sommet H (dont deux sont nettement plus longs que l'itinéraire choisi dans ce cas). Quel que soit l'itinéraire choisi, on peut passer du premier sommet au second, sauf que l'itinéraire peut être tortueux.

Chapitre 11

Obtenir la bonne page Web

DANS CE CHAPITRE

- » Pourquoi il est difficile de trouver ce que l'on veut sur Internet
 - » Les problèmes qui peuvent être résolus grâce à PageRank
 - » Appliquer l'algorithme PageRank au téléport
 - » Comment évolue l'utilisation de PageRank
-

Les chapitres qui précèdent étudient les graphes en détail. L'Internet est un des exemples les plus intéressants, en raison de son étendue et de sa complexité. Ce chapitre permet une meilleure compréhension des algorithmes de base pour le parcours des graphes et l'extraction de structures utiles (comme les regroupements et les communautés), et conclut l'étude sur les graphes par une présentation de l'algorithme PageRank qui a révolutionné la vie des gens tout autant que l'Internet, puisqu'il l'a rendu exploitable. PageRank, ce n'est pas seulement ce qui fait fonctionner Google et bien d'autres moteurs de recherche, c'est aussi un moyen subtil de tirer d'une structure des informations latentes comme la pertinence, l'importance et la réputation d'une page ou d'un site.

Dans les bibliothèques, ce sont les catalogues et les bibliothécaires qui vous facilitent la recherche d'un texte ou d'ouvrages sur un sujet donné. Les livres ne sont pas tous pareils : certains sont meilleurs que d'autres, certains présentent une information plus claire, ou plus complète. Si les recommandations des spécialistes

indiquent qu'un ouvrage est une source fiable, c'est parce qu'elles apparaissent souvent dans d'autres ouvrages sous forme de citations. Or, sur l'Internet, de telles références croisées n'existaient pas dans les premiers temps. C'est la présence de certains mots dans le titre ou dans le corps de texte qui suggérait l'intérêt de telle ou telle page Web. C'était pratiquement comme si l'on devait juger un ouvrage d'après son titre et le nombre de mots qu'il contient.

L'algorithme PageRank a changé tout cela en exploitant les liens sur les pages en guise de recommandations, à l'image des références citées par les spécialistes. L'ampleur croissante de l'Internet joue aussi son rôle dans le succès de cet algorithme. Les bons signaux sont faciles à trouver et se distinguent du bruit par leur apparition régulière. Le bruit, aussi gênant soit-il, est occasionnel par nature. Plus le réseau est étendu, plus on a des chances d'obtenir de bons signaux pour un algorithme intelligent comme PageRank.

Un moteur de recherche pour avoir le monde entier

L'Internet est devenu indispensable à beaucoup de gens, dans leur vie privée ou professionnelle. Le réseau Internet est constitué (entre autres) de pages interconnectées et de sites qui sont accessibles par domaine, chaque domaine étant constitué de pages et d'hyperliens qui relient les éléments d'un site et qui relient chaque site à d'autres sites. Des services et des savoir-faire sont disponibles un peu partout sur l'Internet, à condition de savoir précisément où les trouver. L'accès au Web serait inenvisageable sans les moteurs de recherche, ces sites qui vous permettent de trouver tout ce que vous voulez en utilisant une simple requête.

Rechercher des données sur

l'Internet

D'une taille estimée à près de 50 milliards de pages (<http://www.worldwidewebsize.com/>), le Web n'est pas facile à représenter. Des études le décrivent comme un graphe en forme de nœud papillon (voir <http://www.immorlica.com/socNet/broder.pdf> et <http://vigna.di.unimi.it/ftp/papers/GraphStructureRevisited.pdf>). Il est principalement constitué d'un noyau interconnecté et d'autres composantes reliées à ce noyau. Cependant, certaines parties sont tout à fait inaccessibles. Dans le monde réel, il est possible de se rendre partout (mais pour cela, il faut parfois traverser les océans). Sur le Web, il n'est pas possible d'atteindre tous les sites simplement en suivant la structure : certaines parties ne sont pas facilement accessibles (elles sont déconnectées ou vous n'êtes pas du bon côté pour les atteindre). Si vous voulez trouver quelque chose sur le Web, même si le temps n'est pas un problème, il vous faut un index.

Comment trouver les bonnes données

Trouver les bonnes données est un problème qui se pose depuis les débuts de l'Internet, mais les premiers moteurs de recherche ne sont apparus que dans les années quatre-vingt-dix. On s'en était peu préoccupé auparavant car d'autres solutions, comme les simples listes de domaines ou les catalogues de sites spécialisés, faisaient l'affaire. Ce n'est que lorsque ces solutions ont cessé d'être adaptées en raison de la croissance rapide de l'Internet que sont apparus des moteurs de recherche comme Lycos, Magellan, Yahoo, Excite, Inktomi et Altavista.

Tous ces moteurs de recherche fonctionnent grâce à un logiciel spécialisé qui explore le réseau de façon autonome à l'aide de listes de domaines et de tests des hyperliens rencontrés sur les pages visitées. Ces *robots* explorent chaque nouveau lien selon un processus appelé l'*indexation*. Les robots d'indexation, parfois

appelés araignées du Web, sont des éléments logiciels qui lisent les pages comme du texte simple (ils ne comprennent pas les images et autres contenus non textuels).

Les premiers moteurs de recherche parcouraient le réseau Internet, recueillaient l'information collectée par les robots d'indexation et la traitaient de manière à créer des index inversés. Les index permettaient de retrouver les pages en fonction des mots qu'elles contenaient. Quand on lançait une requête, les index inversés relevaient toutes les pages contenant les termes de la requête et engendraient un classement des pages qui était ensuite présenté comme résultat (une liste de pages ordonnées, depuis celle présumée la plus utile jusqu'à celle présumée la moins utile).

Le classement était simpliste, sachant qu'il se fondait souvent sur la fréquence des mots-clés sur les pages ou sur le fait que ces mots-clés apparaissaient dans le titre ou l'en-tête des pages. Parfois, les mots-clés avaient même un classement plus élevé lorsqu'ils étaient mélangés ou regroupés. À l'évidence, ces techniques simples d'indexation et de classement permettaient à certains utilisateurs de l'Internet d'en tirer profit en utilisant divers trucs :

- » **Les spammeurs** employaient leurs talents à remplir les résultats de recherche de pages inintéressantes et parsemées de publicités.
- » **Les techniques d'optimisation des moteurs de recherche du « chapeau noir**
- » **(Black Hat SEO)** étaient utilisées par ceux qui tiraient parti de leur connaissance des moteurs de recherche pour faire en sorte que ceux-ci attribuent le meilleur classement à des pages qu'ils manipulaient, malgré leur mauvaise qualité. Malheureusement, ces problèmes persistent, car même les moteurs de recherche les plus avancés ne sont pas entièrement protégés contre les individus susceptibles de truquer le système en vue d'obtenir un meilleur classement. L'algorithme PageRank peut résister aux anciennes

techniques de spam et de « chapeau noir », mais il n'est pas une panacée.



Il est essentiel de ne pas confondre Black Hat SEO et White Hat SEO (généralement appelé simplement SEO). Les utilisateurs de White Hat SEO sont des professionnels qui exploitent leur connaissance des moteurs de recherche pour mieux donner la priorité aux pages valides et utiles, de façon légale et éthique.

L'émergence de ces acteurs et la possibilité de manipuler les résultats des moteurs de recherche ont rendu nécessaire la mise en place de meilleurs algorithmes de classement dans les moteurs de recherche. C'est ainsi que l'algorithme PageRank s'est imposé.

DES URL AVEC DES EXTENSIONS .PDF

Les URL qu'on rencontre dans ce livre ont souvent une extension .pdf. Quand on essaie d'ouvrir le lien, on peut voir apparaître un message d'alerte du navigateur indiquant que le fichier .pdf pourrait contenir un virus. En effet, un fichier .pdf peut très bien contenir un virus (voir <http://security.stackexchange.com/questions/64052/can-a-pdf-file-contain-a-virus>). Néanmoins, les liens vers des fichiers .pdf proposés dans ce livre ont peu de chances d'en contenir. Vous pouvez les télécharger sans risque et utiliser ensuite un scanner pour en vérifier le contenu. Avec les fichiers comme avec tout contenu en ligne, mieux vaut prévenir que guérir. Si jamais un des fichiers .pdf référencés dans ce livre contenait un virus, prévenez-nous en nous écrivant à l'adresse John@JohnMuellerBooks.com et contactez l'administrateur du site hébergeur.

Expliquer l'algorithme PageRank

L'algorithme PageRank doit son nom à Larry Page, cofondateur de Google. Il a fait sa première apparition publique en 1998 dans un article de Sergey Brin et Larry Page intitulé « The Anatomy of

a LargeScale Hypertextual Web Search Engine », publié par le journal *Computer Networks and ISDN Systems* (<http://ilpubs.stanford.edu:8090/361/1/1998-8.pdf>). À cette époque, Brin et Page étaient tous les deux doctorants et cet algorithme, le fondement même de la technologie de recherche de Google, était initialement un projet de recherche à l'université de Stanford.

En termes simples, *PageRank* attribue des scores aux sommets d'un graphe de telle sorte que plus le score attribué à un sommet est élevé, plus ce sommet est considéré comme important dans le graphe. Déterminer l'importance d'un sommet dans un graphe qui représente le réseau Internet signifie évaluer la pertinence de chaque page des résultats de la requête, afin de mieux servir les utilisateurs qui recherchent un contenu intéressant.

Une page est une bonne réponse à une requête si elle correspond aux critères de cette requête et si elle occupe une place importante dans le système d'hyperliens qui relie les pages les unes aux autres. Le principe est que la « toile » est ce qu'en font les utilisateurs, et que si une page est importante sur le réseau, c'est pour une bonne raison (la qualité et l'autorité du contenu de la page sont évaluées en fonction de son importance dans le réseau des hyperliens).

Comprendre le raisonnement qui sous-tend l'algorithme PageRank

En 1998, quand Brin et Page étaient encore étudiants à Stanford, la qualité des résultats de recherche était un problème pour tout utilisateur du Web. Les moteurs de recherche traditionnels étaient confrontés à une structure du réseau toujours croissante (la prochaine partie de ce livre traite des problèmes de mise à l'échelle des algorithmes et de la manière de les faire fonctionner avec les jeux de données volumineux) et à toute une foule de spammeurs.

En l'occurrence, il s'agit non pas des spammeurs du courrier électronique (ceux qui vous envoient des courriels non désirés dans votre boîte de réception), mais des spammeurs du Web (ceux qui savent combien il est important du point de vue économique que les pages arrivent en tête des résultats des recherches). Les membres de ce groupe ont mis au point des techniques élaborées et insidieuses pour truquer les résultats des recherches :

- » **Le bourrage de mots-clés** consiste à abuser de certains mots-clés dans une page pour faire croire au moteur de recherche que cette page est réellement consacrée au sujet qu'indiquent les mots-clés.
- » **Le texte invisible** consiste à copier le contenu d'une page qui arrive en tête des résultats sur une page différente en utilisant la même couleur pour les caractères et le fond. Le contenu ainsi copié est invisible pour les utilisateurs, mais pas pour les robots du moteur de recherche (lesquels n'examinent que les données textuelles) ni pour ses algorithmes. La page contenant le texte invisible obtient ainsi un aussi bon classement que la page copiée.
- » **Le cloaking** est une variante plus élaborée du texte invisible. Plutôt que du texte, ce sont des scripts ou des images qui fournissent aux robots des moteurs de recherche un contenu différent de celui que les utilisateurs voient.

Ce sont les trucs qu'utilisent les spammeurs du Web pour tromper les moteurs de recherche et faire en sorte qu'un bon classement soit affecté même à des pages dont le contenu est mauvais, ou dans le meilleur des cas, trompeur. Tout cela n'est pas sans conséquences. Supposons, par exemple, qu'un utilisateur qui recherche des informations concernant la recherche universitaire obtienne de la publicité commerciale, ou un autre contenu inapproprié. De nombreux utilisateurs ont eu la déception d'obtenir trop souvent des pages qui n'avaient rien à voir avec ce qu'ils cherchaient, si bien qu'ils devaient reformuler leurs requêtes, passer du temps à faire un tri des informations des pages de résultats et gaspiller leur énergie à isoler les bonnes références

des mauvaises. Des chercheurs et des spécialistes, confrontés à ce problème de spam et craignant que le Web cesse de se développer, les utilisateurs ayant des difficultés à trouver ce qu'ils recherchent, ont alors commencé à chercher des solutions possibles.

Alors que Brin et Page travaillaient à mettre au point leur algorithme, d'autres idées prenaient forme et donnaient lieu à des développements en parallèle. Ce fut le cas notamment d'Hyper Search, de Massimo Marchiori, le premier à avoir souligné l'importance des liens Internet comme facteur à considérer dans le cadre d'une recherche, pour déterminer l'importance à accorder à une page Web : <https://www.w3.org/People/Massimo/papers/WWW6/>. Une autre solution intéressante est le projet de moteur de recherche HITS (*Hypertext-Induced Topic Search*), également basé sur la structure des liens Internet et développé par Jon Kleinberg, un jeune chercheur travaillant chez IBM Almaden, dans la Silicon Valley. Il est intéressant de noter que HITS classe les pages sous forme de *hubs* (un hub est une page comportant un grand nombre de liens vers des pages de référence) et d'*autorités* (pages considérées comme étant des pages de référence compte tenu de nombreux liens provenant des hubs), ce que PageRank ne fait pas explicitement, mais il le fait implicitement dans ses calculs (<http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Le>



Le moment venu, il arrive souvent que la même idée ou une idée similaire germe en différents endroits. Les scientifiques échangent leurs idées, ou bien développent des projets analogues de façon totalement indépendante (voir l'histoire du mathématicien japonais Seki Takakazu, <http://www-history.mcs.st-andrews.ac.uk/history/Biographies/Seki.html>, contemporain de Newton, Leibniz et Bernoulli, qui de façon indépendante avait établi des résultats semblables à ceux de ces mathématiciens européens). En 1998, lorsque Brin et Page ont pris congé de l'université de Stanford pour fonder une société, ils ont été les seuls à développer un moteur de recherche fondé sur leur algorithme. Ils ont alors consacré leur temps à faire en sorte que

cet algorithme puisse fonctionner avec plus d'un milliard de pages Web.

Comment fonctionne PageRank ?

Ce qui a changé avec PageRank, c'est qu'un index inversé des termes ne suffit plus pour déterminer si une page répond ou non à la demande d'information de l'utilisateur. La correspondance entre les mots, entre la requête et le texte de la page (ou la correspondance sémantique, voir l'étude à la fin de ce chapitre), est une condition préalable, mais non suffisante, car les hyperliens sont nécessaires pour déterminer si la page présente un contenu de qualité et constitue une référence sur le sujet concerné.

Dans l'examen des sites, il est important de faire la distinction entre les liens entrants et les liens sortants, tandis que les liens internes au site ne doivent pas être pris en compte. Les liens qui apparaissent sur une page sont des liens sortants s'ils conduisent à une page située sur un autre site. Les liens qui amènent l'utilisateur sur votre page depuis une page d'un autre site sont des liens entrants (appelés aussi liens retour). En tant que créateur de la page, vous utilisez les liens sortants pour enrichir le contenu de votre page par des informations supplémentaires. Sur votre page, vous n'insérerez sans doute pas des liens au hasard (ni des liens pointant vers un contenu mauvais ou inutile), car ce serait préjudiciable à la qualité de votre page. De même que vous allez plutôt créer des liens pointant vers un contenu pertinent et de qualité, d'autres créateurs de pages inséreront sur leurs pages des liens pointant vers votre page si celle-ci est intéressante et de bonne qualité.

C'est une chaîne de confiance. Un hyperlien est une façon de cautionner ou de recommander une page. Les liens entrants vous montrent que d'autres créateurs de pages vous font confiance, et la confiance est partagée lorsque vous ajoutez vous-même à vos pages des liens sortants qui pointent vers les pages des autres.

Mettre en œuvre PageRank

Représenter cette chaîne de confiance implique une détermination mathématique simultanée du degré d'autorité de votre page, mesuré par le nombre de liens entrants, et de l'importance que votre page accorde aux autres pages, mesurée par le nombre de liens sortants. Ces calculs peuvent être effectués de deux manières :

- » **La simulation** : Il s'agit de simuler le comportement d'un utilisateur qui surfe sur le Web de façon aléatoire (un *internaute aléatoire*). Il faut pour cela reproduire la structure du Web, avant de lancer la simulation.
- » **Le calcul matriciel** : Il s'agit de reproduire le comportement d'un utilisateur qui surfe de façon aléatoire sur le Web, en se servant d'une *matrice creuse* (une matrice dans laquelle la plupart des valeurs sont nulles) qui reproduit la structure du Web. Cette méthode fait appel à des opérations matricielles, comme expliqué au [Chapitre 5](#), et à une série de calculs aboutissant à un résultat par approximations successives.

Le calcul matriciel utilisé pour PageRank est plus abstrait mais nécessite moins d'instructions de programmation, et il est facile de le mettre en application à l'aide de Python. Vous pouvez faire fonctionner l'algorithme PageRank sur des sites réels en utilisant un vérificateur automatique de PageRank comme <http://checkpagerank.net/index.php>. Malheureusement, le programme produit parfois des résultats incorrects pour les nouveaux sites, qui n'ont pas encore été explorés et indexés correctement. Cela peut vous donner une idée du fonctionnement de PageRank dans la pratique.

Implémenter un script Python

PageRank est une fonction qui attribue un score à chaque sommet d'un graphe (plus ce score est élevé, plus le sommet est important). Le score attribué à une page Web représente la probabilité qu'un internaute aléatoire visite cette page. Les probabilités sont exprimées par un nombre compris entre 0.0 et 1.0 et dans la mesure du possible, lorsque l'on représente la probabilité d'être sur un site particulier parmi l'ensemble des sites accessibles, la somme de toutes les probabilités associées aux pages Web doit être égale à 1.0.



Il existe un certain nombre de versions différentes de PageRank, la recette changeant un peu à chaque fois pour s'adapter au graphe à traiter. L'exemple choisi dans cette section est la version originale pour le Web, telle qu'elle est présentée dans l'article de Brin et Page mentionné précédemment ainsi que dans l'article « PageRank : Bringing Order to the Web » (<http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf>).

Dans cet exemple, on crée trois réseaux Internet différents constitués de six sommets (pages Web). Le premier est un réseau qui fonctionne bien, tandis que les deux autres posent des problèmes qu'un internaute aléatoire est susceptible de rencontrer en raison de la structure du Web ou de l'activité d'un spammeur du Web. Cet exemple utilise aussi les commandes NetworkX évoquées au [Chapitre 8](#) (vous retrouverez ce code dans le fichier téléchargeable A4D ; 11 ; PageRank.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
Graph_A = nx.DiGraph()
Graph_B = nx.DiGraph()
Graph_C = nx.DiGraph()
Nodes = range(1,6)
Edges_OK = [(1,2),(1,3),(2,3),(3,1),(3,2),(3,4),(4,5),
            (4,6),(5,4),(5,6),(6,5),(6,1)]
Edges_dead_end = [(1,2),(1,3),(3,1),(3,2),(3,4),(4,5),
                  (4,6),(5,4),(5,6),(6,5),(6,1)]
Edges_trap = [(1,2),(1,3),(2,3),(3,1),(3,2),(3,4),
```



```
(4, 5),
      (4, 6), (5, 4), (5, 6), (6, 5)]
Graph_A.add_nodes_from(Nodes)
Graph_A.add_edges_from(Edges_OK)
Graph_B.add_nodes_from(Nodes)
Graph_B.add_edges_from(Edges_dead_end)
Graph_C.add_nodes_from(Nodes)
Graph_C.add_edges_from(Edges_trap)
```

Ce code fait apparaître sur l'écran le premier réseau, celui qui fonctionne bien ([Figure 11-1](#)).

```
np.random.seed(2)
pos=nx.shell_layout(Graph_A)
nx.draw(Graph_A, pos, arrows=True, with_labels=True)
plt.show()
```



Tous les sommets sont reliés entre eux. C'est l'exemple d'un graphe fortement connecté, sans sommets isolés ni enclaves en cul-de-sac. Un internaute aléatoire peut y naviguer librement sans jamais être obligé de s'arrêter, tout sommet permettant de gagner un autre sommet. Dans la représentation NetworkX d'un graphe orienté, il n'y a pas de flèches mais le sens d'un arc est représenté par une ligne plus épaisse à l'arrivée à un sommet. L'internaute peut se rendre, par exemple, du sommet 4 au sommet 6, sachant que la ligne tracée entre ces deux sommets devient plus épaisse à l'arrivée au sommet 6. En revanche, il ne peut pas se rendre du sommet 6 au sommet 4 puisqu'aux abords du sommet 4, la ligne qui relie ces deux sommets est étroite.

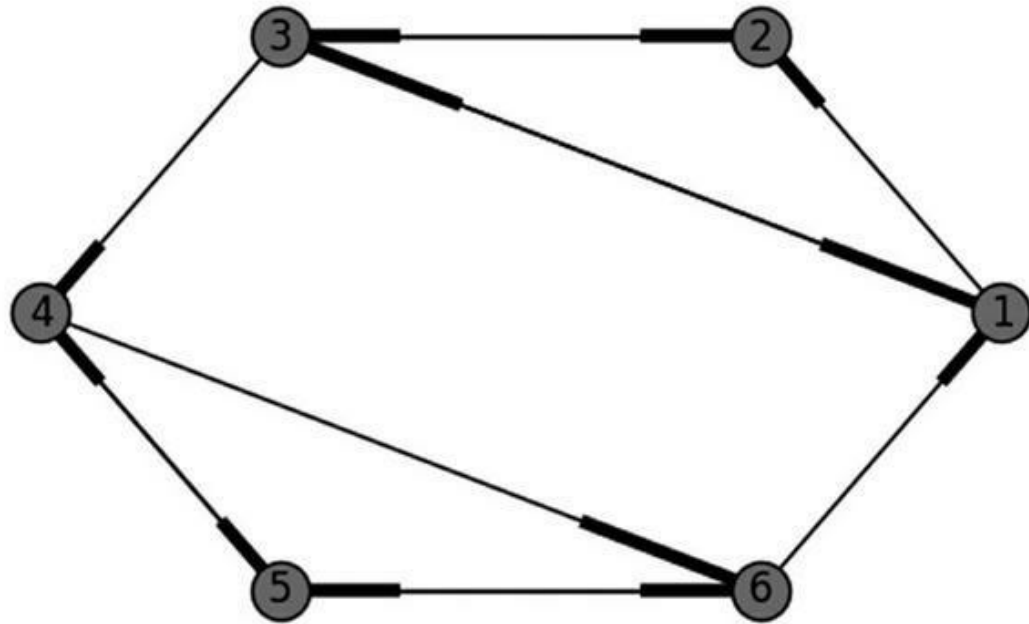


FIGURE 11-1 Un réseau très connecté.

Le second graphe n'est pas fortement connecté. Il présente un piège pour l'internaute aléatoire car le second sommet n'a pas de liens sortants. Un utilisateur qui visite cette page risque donc de devoir s'arrêter à ce sommet, faute de trouver une issue. Ce n'est pas un événement inhabituel, compte tenu de la structure du Web, mais ce peut être aussi un artéfact résultant d'un spammeur ayant créé une fabrique de spams avec de nombreux liens pointant sur un site sans issue afin de piéger les internautes. La [Figure 11-2](#) représente l'output du code suivant, qui a été utilisé pour afficher ce graphe :

```
np.random.seed(2)
pos=nx.shell_layout(Graph_B)
nx.draw(Graph_B, pos, arrows=True, with_labels=True)
plt.show()
```

Une autre situation qui peut se produire naturellement ou résulter de l'activité d'un spammeur est le piège à robots. Cette fois, ce n'est pas une page unique qui piège l'utilisateur, mais un site fermé sans lien vers un réseau de pages externes. La [Figure 11-](#)

3 représente l'output du code suivant, qui a été utilisé pour afficher ce graphe :

```
np.random.seed(2)
pos=nx.shell_layout(Graph_C)
nx.draw(Graph_C, pos, arrows=True, with_labels=True)
plt.show()
```



On parle de piège à robots parce que les spammeurs ont conçu ce dispositif pour piéger les logiciels des moteurs de recherche dans une boucle et leur faire croire que les seuls sites Web à prendre en considération sont ceux appartenant à ce réseau fermé.

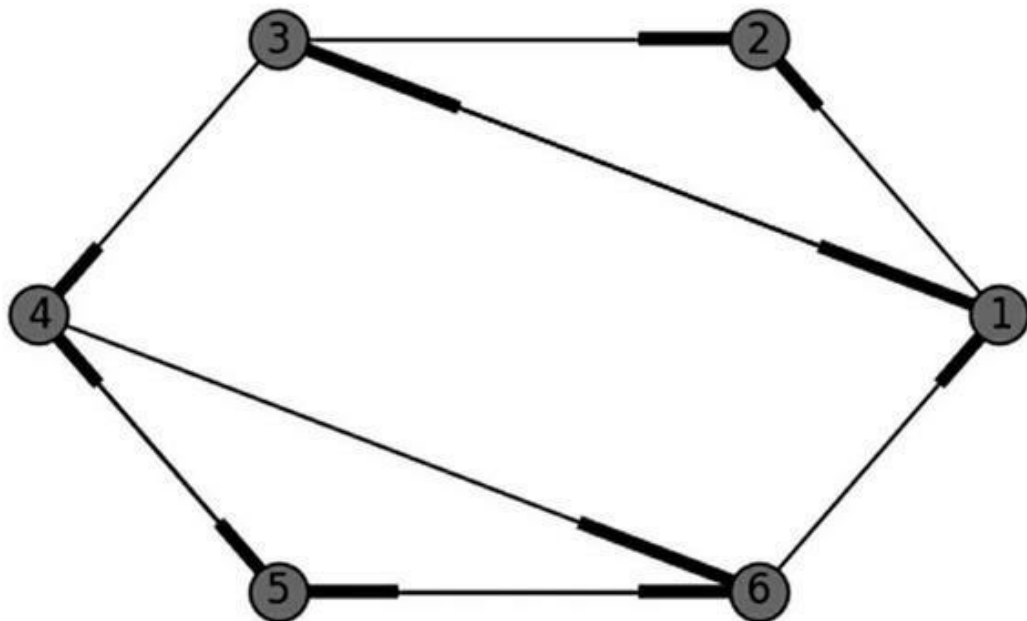


FIGURE 11-2 Une voie sans issue.

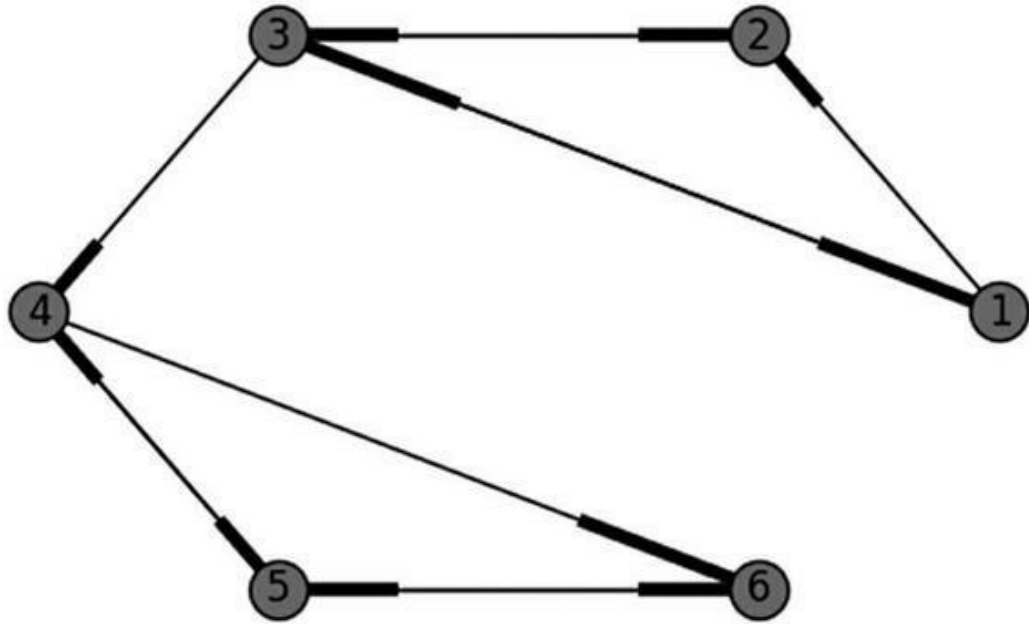


FIGURE 11-3 Un piège à robots.

Procéder à une implémentation naïve

On peut extraire la structure d'un graphe élaboré à l'aide de Python et NetworkX et le représenter par une matrice de transition :

- » **Les colonnes** représentent le sommet sur lequel se trouve l'internaute.
- » **Les lignes** représentent la probabilité que l'internaute visite d'autres sommets grâce aux liens sortants.

Sur le réseau réel, la matrice de transition qui alimente l'algorithme PageRank est créée par l'exploration ininterrompue des liens par les robots d'indexation.

```
def initialize_PageRank(graph):  
    nodes = len(graph)  
    M = nx.to_numpy_matrix(graph)  
    outbound = np.squeeze(np.asarray(np.sum(M,
```

```

axis=1)))
    prob_outbound = np.array(
        [1.0/count
         if count>0 else 0.0 for count in outbound])
    G = np.asarray(np.multiply(M.T, prob_outbound))
    p = np.ones(nodes) / float(nodes)
    if np.min(np.sum(G,axis=0)) < 1.0:
        print ('Attention : G est sous-stochastique')
    return G, p

```

Le code Python crée la fonction `initialize_PageRank` qui extrait la matrice de transition et le vecteur initial des scores PageRank par défaut.

```

G, p = initialize_PageRank(Graph_A)
print (G)

[[ 0.  0.  0.33333333  0.  0.  0.5 ]
 [ 0.5  0.  0.33333333  0.  0.  0. ]
 [ 0.5  1.  0.  0.  0.  0. ]
 [ 0.  0.  0.33333333  0.  0.5  0. ]
 [ 0.  0.  0.  0.5  0.  0.5 ]
 [ 0.  0.  0.  0.5  0.5  0. ]]

```

La matrice de transition `G` affichée correspond au réseau de la [Figure 11-1](#). Chaque colonne représente un sommet de la séquence des sommets 1 à 6. La troisième colonne, par exemple, représente le sommet 3. Les lignes de cette colonne représentent les connexions avec les autres sommets (liens sortants vers les sommets 1, 2 et 4) et les valeurs qui définissent la probabilité qu'un internaute aléatoire utilise ces liens sortants (: 1/3, 1/3, 1/3).



La diagonale de la matrice est toujours constituée de zéros, sauf lorsqu'une page comporte un lien sortant pointant vers elle-même (ce qui reste possible).

La matrice comporte plus de zéros que de valeurs non nulles. C'est aussi le cas dans la réalité, car comme le montrent des estimations, chaque site ne comporte en moyenne que dix liens sortants. Sachant qu'il existe plus d'un milliard de sites, les valeurs non nulles dans une matrice de transition représentant le

Web sont en nombre très réduit. En l'occurrence, il est utile de recourir à une liste d'adjacence (voir [Chapitre 8](#)) en guise de structure de données pour représenter les données sans gaspiller de l'espace disque ou mémoire avec des valeurs nulles :

```
from scipy import sparse
sG = sparse.csr_matrix(G)
print (sG)
```

```
(0, 2) 0.333333333333
(0, 5) 0.5
(1, 0) 0.5
(1, 2) 0.333333333333
(2, 0) 0.5
(2, 1) 1.0
(3, 2) 0.333333333333
(3, 4) 0.5
(4, 3) 0.5
(4, 5) 0.5
(5, 3) 0.5
(5, 4) 0.5
```

Dans cet exemple, on compte 12 liens sur 30 liens possibles (sans compter les liens du site courant vers lui-même). Une autre particularité de la matrice de transition est que la somme de chaque colonne doit être égale à 1,0. Si elle est inférieure à 1,0, la matrice est sous-stochastique (ce qui signifie que les données de la matrice ne représentent pas correctement les probabilités, sachant que la somme des probabilités doit toujours être égale à 1,0) et ne peut pas fonctionner de façon satisfaisante avec les estimations de PageRank.

Le point G s'accompagne du vecteur p, l'estimation initiale du score total de PageRank, également distribué parmi les sommets. Dans cet exemple, le PageRank total vaut 1,0 (la probabilité qu'un internaute aléatoire soit sur le réseau est de 100 %), et sa distribution est de 1/6 sur chacun des six sommets :

```
print(p)

[ 0.16666667 0.16666667 0.16666667 0.16666667
 0.16666667 0.16666667]
```

Pour estimer le PageRank, on part de l'estimation initiale pour un sommet dans le vecteur p , on la multiplie par la colonne correspondante dans la matrice de transition, et l'on détermine ainsi la part (l'autorité) du PageRank à attribuer aux autres sommets. La répétition de cette opération pour tous les sommets permet de connaître la distribution du PageRank entre les sommets, compte tenu de la structure du réseau. Le calcul peut être réalisé sous la forme d'une multiplication du vecteur par la matrice :

```
print(np.dot(G,p))  
  
[ 0.13888889 0.13888889 0.25 0.13888889  
 0.16666667 0.16666667]
```

Après la première multiplication du vecteur par la matrice, on obtient une autre estimation du PageRank qui est alors utilisée pour la redistribution entre les sommets. La redistribution répétée stabilise l'estimation du PageRank (les résultats ne varient pas) et l'on obtient le score dont on avait besoin. En utilisant une matrice de transition constituée des probabilités et en procédant à l'estimation par approximations successives à l'aide de la multiplication matrice-vecteur, on obtient les mêmes résultats qu'avec une simulation sur ordinateur avec un internaute aléatoire :

```
def PageRank_naive(graph, iters = 50):  
    G, p = initialize_PageRank(graph)  
    for i in range(iters):  
        p = np.dot(G,p)  
    return np.round(p,3)  
  
print(PageRank_naive(Graph_A))  
  
[ 0.154 0.154 0.231 0.154 0.154 0.154]
```

La nouvelle fonction `PageRank_naive` recouvre toutes les opérations déjà évoquées et produit un vecteur de probabilités (le score de PageRank) pour chaque sommet dans le réseau. Le troisième sommet apparaît comme le plus important.

Malheureusement, cette même fonction ne fait pas l'affaire avec les deux autres réseaux :

```
print(PageRank_naive(Graph_B))  
Attention : G est sous-stochastique  
[ 0. 0. 0. 0. 0. 0.]  
  
print(PageRank_naive(Graph_C))  
[ 0. 0. 0. 0.222 0.444 0.333]
```

Dans le premier cas, les probabilités semblent désertir le réseau : c'est l'effet d'un site Web sans issue et de la matrice de transition sous-stochastique résultante. Dans le second cas, la moitié inférieure du réseau monopolise indûment toute l'importance, au détriment de la partie supérieure qui ne compte plus.

Vers la téléportation

Les voies sans issue (*rank sinks*) et les pièges à robots (*boucles*) sont des situations courantes sur Internet, qui résultent des choix des utilisateurs et de l'activité des spammeurs. Cependant, ce problème peut facilement être résolu en faisant passer l'internaute aléatoire directement à un autre sommet par un saut aléatoire (en le *téléportant*, comme le font ces appareils de science-fiction qui vous déplacent instantanément d'un endroit à un autre). En théorie, l'internaute se lassera à un moment donné et se sortira d'une situation sans issue. Mathématiquement, on définit une valeur alpha représentant la probabilité que l'internaute poursuive sa navigation aléatoire sur le graphe. La valeur alpha redistribue la probabilité d'être sur un sommet donné indépendamment de la matrice de transition.



La valeur initialement suggérée par Brin et Page pour alpha (aussi appelée facteur d'amortissement) est de 0,85, mais vous pouvez la changer selon vos besoins. Pour le Web, l'idéal est que cette valeur soit comprise entre 0,8 et 0,9 et si vous voulez savoir pourquoi, consultez la page <https://www.cise.ufl.edu/~adobra/DaMn/talks/damn05-santini.pdf>.

En moyenne, plus la valeur alpha est petite, plus le parcours de l'internaute sur le réseau est court avant qu'il reparte ailleurs.

```
def PageRank_teleporting(graph, iters = 50,
alpha=0.85,
    rounding=3):
    G, p = initialize_PageRank(graph)
    u = np.ones(len(p)) / float(len(p))
    for i in range(iters):
        p = alpha * np.dot(G,p) + (1.0 - alpha) * u
    return np.round(p / np.sum(p), rounding)
print('Graphe A:', PageRank_teleporting(Graph_A,
    rounding=8))
print('Graphe B:', PageRank_teleporting(Graph_B,
    rounding=8))
print('Graphe C:', PageRank_teleporting(Graph_C,
    rounding=8))
```

```
Graphe A: [ 0.15477863 0.15346061 0.22122243
0.15477863
    0.15787985 0.15787985]
Attention : G est sous-stochastique
Graphe B: [ 0.16502904 0.14922238 0.11627717
0.16502904
    0.20222118 0.20222118]
Graphe C: [ 0.0598128 0.08523323 0.12286869 0.18996342
0.30623677 0.23588508]
```

Après avoir appliqué les modifications à une nouvelle fonction, PageRank_tele-porting, vous pouvez obtenir des estimations similaires pour le premier graphe et des estimations bien plus réalistes (et plus utiles) pour le deuxième et le troisième graphes, sans tomber dans les pièges des culs-de-sac et autres voies sans issue. On remarque que cette fonction est équivalente à celle proposée par NetworkX : http://networkx.readthedocs.io/en/networkx-1.11/reference/generated/networkx.algorithms.link_analysis.pagerank.html

```
nx.pagerank(Graph_A, alpha=0.85)
```

```
{1: 0.15477892494151968,
2: 0.1534602056628941,
3: 0.2212224378270561,
4: 0.15477892494151968,
```

5: 0.1578797533135051,
6: 0.15787975331350507}

Voir de plus près à quoi ressemble un moteur de recherche

Bien qu'il ne rende compte que de la structure des hyperliens de l'Internet, PageRank révèle dans quelle mesure une page peut faire autorité. PageRank n'est cependant pas la seule composante de Google. L'algorithme est une base solide pour toutes les requêtes, et il a joué initialement un rôle essentiel pour établir la réputation de fiabilité de Google en tant que moteur de recherche. Aujourd'hui, PageRank est simplement un des divers facteurs de positionnement qui interviennent dans le traitement d'une requête.

D'après les sources spécialisées dans le savoir-faire en matière d'optimisation des moteurs de recherche, plus de 200 facteurs contribuent aux résultats fournis par Google. Pour connaître les autres facteurs pris en compte par Google, consultez la liste sur la page <https://moz.com/search-ranking-factors> (une liste dressée par la société américaine MOZ). Vous pouvez également télécharger les rapports annuels de Searchmetrics, une société berlinoise spécialisée dans les logiciels d'optimisation des moteurs de recherche : <http://www.searchmetrics.com/knowledge-base/ranking-factors/>.

Il faut aussi considérer que l'algorithme de Google a fait l'objet de nombreuses mises à jour et qu'il est aujourd'hui constitué d'un ensemble d'algorithmes différents qui ont reçu chacun un nom fantaisie (Caffeine, Panda, Penguin, Hummingbird, Pigeon, Mobile Update). Ces mises à jour ont souvent remis en cause les classements précédents et elles étaient motivées par le besoin de contrer les techniques de spamming ou par le souci de rendre la navigation sur le Web plus pratique pour les utilisateurs. (La *Mobile Update*, par exemple, a conduit un grand nombre de sites à rendre leurs interfaces compatibles avec les téléphones mobiles).

Envisager d'autres utilisations de PageRank

Si PageRank permet d'obtenir de meilleurs résultats de recherche, son domaine d'application ne se limite pas à Google ni aux moteurs de recherche. Vous pouvez utiliser PageRank chaque fois que le problème à résoudre peut être présenté sous forme de graphe. Il vous suffit de modifier et de paramétrer l'algorithme pour l'adapter à vos besoins. L'université Cornell a recensé d'autres utilisations possibles de PageRank dans différents secteurs (<https://blogs.cornell.edu/info2040/2014/11/03/more-than-just-a-web-search-algorithm-googles-pagerank-in-non-internet-contexts/>), et des rapports surprenants font état de l'emploi avec succès de cet algorithme en biologie computationnelle (<https://www.wired.com/2009/09/googlefoodwebs/>). Quand vous créez une « téléportation » pour les sommets que vous voulez explorer, l'algorithme fait merveille dans des applications comme les suivantes :

- » **Détection des fraudes** : Quand l'algorithme révèle des liens inattendus entre certaines personnes et entre certains faits.
- » **Recommandation de produits** : Quand l'algorithme permet de proposer des produits aux destinataires dont les affinités sont telles qu'ils sont particulièrement susceptibles de les trouver intéressants.

Au-delà du paradigme de PageRank

Ces dernières années, Google n'a pas seulement ajouté des facteurs de positionnement qui modifient l'algorithme PageRank original. La compagnie a réalisé d'importants changements pour améliorer l'évaluation du contenu des pages (afin d'éviter que l'algorithme soit abusé par la présence de certains mots-clés) et a

adopté les algorithmes d'intelligence artificielle qui notent la pertinence d'une page dans les résultats de recherche de façon autonome. Compte tenu de ces changements, certains spécialistes ont déclaré que ce n'est plus PageRank qui détermine la position d'une page dans les résultats d'une requête (voir <https://www.entrepreneur.com/article/269574>). La question reste controversée, mais on peut raisonnablement supposer que PageRank continue de jouer un rôle important dans le moteur de recherche de Google en tant que facteur de positionnement, même s'il ne peut plus à lui seul déterminer la présence d'une page dans les meilleurs résultats d'une requête.

À la découverte des requêtes sémantiques

En posant des questions à Google, plutôt que de saisir simplement des chaînes de mots-clés, vous constaterez que le moteur de recherche répond de façon intelligente, comme s'il comprenait véritablement le sens de votre question. En effet, depuis 2012, Google est devenu capable de comprendre les synonymes et les concepts. Après la mise à jour Hummingbird en août 2013 (<http://searchengineland.com/google-hummingbird-172816>), le moteur de recherche est même devenu capable de comprendre les requêtes en mode conversationnel (les demandes formulées comme lorsque l'on s'adresse à quelqu'un) ainsi que la sémantique des requêtes et des contenus des pages.

Depuis cette mise à jour, l'algorithme de Google va au-delà des mots-clés et lève les ambiguïtés dans les intentions de l'utilisateur comme dans la signification du contenu des pages. Désormais, il fonctionne davantage selon une logique sémantique, consistant à identifier le sens des mots du côté de la requête comme du côté des pages Web à afficher. De ce fait, il n'est plus possible de tromper le moteur de recherche en jouant avec les mots-clés. Même indépendamment de l'aide de PageRank, il peut analyser la

façon dont une page est écrite et juger si son contenu justifie son inclusion dans les résultats de la requête.

Utiliser l'intelligence artificielle pour classer les résultats de recherche

PageRank reste au cœur du processus, mais les résultats ont moins de poids en raison de l'introduction de RankBrain, une technologie d'apprentissage machine. D'après certaines sources (voir <https://www.bloomberg.com/news/articles/2015-10-26/google-turning-its-lucrative-web-search-over-to-aima-chines>), cet algorithme d'intelligence artificielle examine désormais toutes les requêtes lancées sur Google et gère directement 15 % du volume des requêtes qu'il reçoit chaque jour. Il traite plus particulièrement :

- » les requêtes ambiguës ou peu claires ;
- » les requêtes exprimées en argot ou en termes familiers ;
- » les requêtes exprimées sous la même forme que si elles faisaient partie d'une conversation avec le moteur de recherche.

RankBrain est encore tenu secret, mais on sait que l'algorithme semble capable de décider, avec bien plus de pertinence que pourrait le faire un être humain, si les contenus d'une page peuvent apparaître dans les résultats d'une recherche donnée. Dans les cas qui sont difficiles à juger, il remplace tous les autres facteurs de positionnement. Nous avons là encore un exemple d'algorithme supplémentaire réduisant le rôle joué par l'algorithme initial PageRank.

PARTIE 4

Dans l'univers des grandes données

DANS CETTE PARTIE...

Interagir avec de vastes jeux de données

Traiter les données en continu, afin de pouvoir utiliser des jeux de données encore plus grands

Exécuter des tâches en parallèle afin d'accélérer le travail de gestion et d'analyse

Coder les données pour éliminer les redondances et assurer la sécurité des données

Compresser et décompresser les données à l'aide de l'algorithme LZW

Chapitre 12

Gérer les grandes données

DANS CE CHAPITRE

- » Comprendre pourquoi les grandes données sont aujourd'hui un élément moteur
 - » Se familiariser à la loi de Moore et à ses implications
 - » Faire le point sur les grandes données et leurs 4 V
 - » Découvrir le moyen de gérer un flux ininterrompu de données
 - » Exploiter l'échantillonnage, le hachage et les aperçus dans le cadre du traitement des données en continu
-

Les grandes données ne sont pas un simple concept utilisé par les entreprises pour proposer de nouvelles techniques de stockage et d'analyse des données. Elles sont une réalité et un élément moteur de l'économie actuelle. Vous avez sans doute déjà rencontré cette expression dans diverses publications spécialisées, et vous vous êtes peut-être demandé ce qu'elle signifiait. Techniquement parlant, les grandes données (*big data*) font référence à de vastes ensembles complexes de données informatiques, dont la dimension est telle (comme leur nom l'indique) qu'il n'est pas possible de simplement augmenter les capacités de stockage sur les ordinateurs ni de compter sur le progrès de la puissance de traitement et de la rapidité des calculs. Les grandes données révolutionnent le stockage et la gestion des données.

Néanmoins, ce stockage considérable et sophistiqué des données n'est pas une chose soudainement apparue. Il a fallu du temps pour mettre au point une technologie permettant de stocker de telles quantités de données. Il a fallu également du temps pour diffuser la technologie de production et de distribution de ces données : ordinateurs, capteurs, téléphones mobiles intelligents, l'Internet et ses services Web. Ce chapitre étudie les raisons qui sous-tendent cette production colossale de données.

Même s'il a fallu du temps pour accumuler de telles quantités de données, c'est l'évolution récente des technologies qui a finalement permis à la population de se rendre compte de l'enjeu considérable qu'elles représentent (quelle que soit leur nature). Depuis plusieurs siècles, l'accent était mis sur la capacité de l'intellect humain à déterminer, à partir d'un petit nombre d'observations précises (petites données), les causes et les forces dont dépendent les événements de la nature.

L'être humain a aussi mis au point une méthode, la méthode scientifique, qui constitue le fondement de la société moderne et qui repose sur le principe de la découverte scientifique. On s'est aperçu un beau jour (non sans une certaine surprise) qu'il était possible de résoudre les problèmes plus vite et de façon plus efficace en cherchant la solution dans une grande quantité de données au lieu de consacrer de longues années à développer et élaborer des théories à l'aide de tests et d'expérimentations bien conçus.

D'abondantes quantités de données ne suffisent pas pour trouver des solutions aux nombreux problèmes qui se posent encore dans notre civilisation. Cependant, disposer du bon algorithme et des données suffisantes, ce qui signifie aujourd'hui des quantités considérables de données, permet de trouver les liens qui peuvent exister entre des milliers d'indices. Les grandes données et les algorithmes donnent accès à de merveilleuses découvertes scientifiques (qui ont des applications pratiques).

Transformer l'énergie électrique en données

En 1965, Gordon Moore, cofondateur d'Intel et de Fairchild Semiconductor (deux géants de la production de composants d'ordinateurs et autres appareils électroniques), déclarait dans un article intitulé « Cramming More Components Onto Integrated Circuits » et publié dans le magazine *Electronics* que le nombre de composants dans les circuits intégrés allait doubler chaque année au cours de la décennie à venir. À cette époque, l'électronique reposait sur les transistors. Pouvoir inclure davantage de transistors dans un circuit sous forme d'un unique composant électronique réunissant les fonctionnalités d'un grand nombre d'éléments (un circuit intégré), c'était pouvoir fabriquer des dispositifs électroniques plus puissants et plus utiles. Ce processus d'*intégration* repose sur la miniaturisation des composants électroniques (rendre un même circuit beaucoup plus petit, ce qui est logique, le même volume devant contenir deux fois plus de circuits que l'année précédente).

Avec la miniaturisation, les appareils électroniques, qui sont le produit final de ce processus, deviennent plus petits ou simplement plus performants. Les ordinateurs actuels, par exemple, ne sont pas nettement plus petits que ceux d'il y a dix ans, mais ils sont beaucoup plus puissants. Il en est de même des téléphones mobiles. Même lorsqu'ils ont les mêmes dimensions que les modèles de la génération précédente, ils sont capables d'exécuter davantage de tâches. D'autres appareils, comme les capteurs, sont simplement plus petits, ce qui permet de les installer n'importe où.

Comprendre les implications de la loi de Moore

Ce que Moore avait énoncé dans ce fameux article s'est révélé vrai pendant longtemps, et c'est ce que, dans l'industrie des semi-conducteurs, on a appelé la loi de Moore (pour plus de détails, voir <http://www.moorelaw.org/>). Pendant dix ans, ce doublement annuel a bien été observé comme prévu. En 1975, Moore a corrigé son énoncé et a parlé d'un doublement tous les deux ans. La [Figure 12-1](#) montre les effets de ce phénomène. Le rythme prévu reste d'actualité, sauf qu'il est généralement considéré aujourd'hui que cette loi cessera de se vérifier à la fin de la décennie (aux environs de 2020). À partir de 2012, on observe une discordance entre la croissance prévue du nombre de transistors intégrés dans un composant pour le rendre plus performant et ce que les fabricants de semi-conducteurs sont capables de réaliser en matière de miniaturisation. En réalité, il existe des barrières physiques à l'accroissement du nombre d'éléments d'un circuit intégré, tant que l'on continue d'utiliser les composants actuels en silicium (néanmoins, les innovations se poursuivront : pour plus de détails, lisez l'article à l'adresse <http://www.nature.com/news/the-chips-are-down-for-moores-law-1.19338>). Par ailleurs, la loi de Moore n'est pas vraiment une loi. Les lois physiques, comme la loi de la gravitation universelle (découverte par Newton, et qui explique pourquoi les objets sont attirés par le sol), se fondent sur diverses sortes de preuves dont l'exactitude a fait l'objet d'une évaluation par les pairs. Or, la loi de Moore n'est rien d'autre qu'une simple observation, ou même un timide objectif que s'est fixé cette industrie (une prophétie autoréalisatrice, en quelque sorte). Dans l'avenir, la loi de Moore risque de ne plus se vérifier car l'industrie adoptera un jour ou l'autre une nouvelle technologie (consistant, par exemple, à fabriquer les composants en utilisant des lasers optiques à la place des transistors : pour plus de détails sur l'ordinateur optique, lire l'article à l'adresse <http://www.extremetech.com/extreme/187746-by-2020-you-could-have-an-exascale-speed-of-light-optical-computer-on-your-desk>). Ce qu'il importe de retenir, c'est que depuis 1965, l'industrie informatique a franchi tous les deux ans environ une nouvelle étape dans le domaine de l'électronique numérique, non sans d'importantes conséquences.

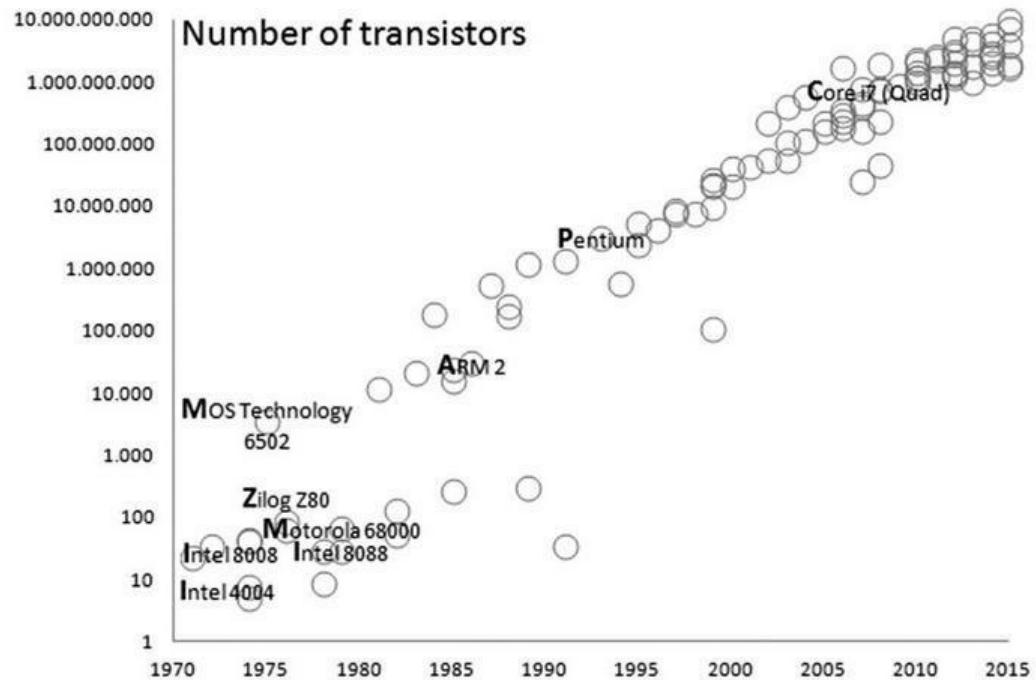


FIGURE 12-1 L'intégration d'un nombre de plus en plus élevé de transistors dans un processeur.



Certains affirment que la loi de Moore ne se vérifie déjà plus. L'industrie des circuits intégrés a tenu promesse jusqu'ici, mais les attentes sont maintenant revues à la baisse. Intel a déjà augmenté le temps d'attente entre une génération de processeurs et la suivante et a annoncé que d'ici cinq ans, la miniaturisation des puces allait atteindre sa limite. Pour en savoir davantage sur cet intéressant sujet, vous pouvez lire un article de la *MIT Technology Review* à l'adresse <https://www.technologyreview.com/s/601441/mooreslaw-is-dead-now-what/>.

La loi de Moore a un effet direct sur les données. Cela commence par les appareils intelligents. Plus les appareils sont intelligents, plus grande est leur diffusion (de nos jours, l'électronique est partout). Plus grande est la diffusion, plus les prix baissent, ce qui entraîne une boucle sans fin, et c'est ainsi qu'aujourd'hui on trouve partout des machines de calcul de forte capacité et de petits capteurs. La mémoire informatique est disponible en abondance,

les supports de données offrent des capacités de stockage considérables, ce qui a pour conséquence une disponibilité généralisée des données, notamment sur les sites Web, des transactions enregistrées et toutes sortes de mesures, d'images numériques et autres types de données qui circulent partout.

Quand on trouve des données partout

Les scientifiques ont commencé à devoir faire face à d'impressionnantes quantités de données bien avant que quelqu'un invente la notion de *grandes données*. Il fut un temps où l'Internet était loin de produire ces vastes séries de données qu'il produit aujourd'hui. Il importe de ne pas oublier que les grandes données ne sont pas simplement un truc commercial trouvé par les fabricants de logiciels et de matériels informatiques, mais une notion fondamentale dans les domaines suivants :

- » **L'astronomie** : Considérons les données transmises par un engin spatial au cours de sa mission (par exemple, la sonde Voyager ou Galileo) et toutes les données reçues des radiotélescopes, qui sont des antennes spécialisées servant à capter les ondes radio émises par les corps astronomiques. Un exemple classique est le projet Search for Extraterrestrial Intelligence (SETI) (<http://www.seti.org/>), qui consiste à rechercher des signaux extraterrestres en étudiant les fréquences radio captées dans l'espace. La quantité de données reçues et la puissance informatique utilisée pour analyser une partie du ciel ne serait-ce que pendant une heure sont impressionnantes (<http://www.setileague.org/askdr/howmuch.htm>). S'il existe des extraterrestres quelque part de ce côté, ils sont très difficiles à repérer (Le film *Contact*, <https://www.amazon.com/exec/obidos/ASIN/B002GHHHKQ20/>, parle de ce qui pourrait se produire si l'homme interceptait effectivement un signal).

PRENDRE EN COMPTE LES ASPECTS POLITIQUES DE CERTAINES LOIS PHYSIQUES

Selon l'identité de votre interlocuteur, la grande question de savoir si une loi résistera à l'épreuve du temps ne sera pas débattue de la même manière. Tout est question de point de vue. Ce livre n'a pas pour but de vous faire adopter un certain point de vue, il s'agit simplement ici de faire état de l'opinion prédominante. Ainsi, par exemple, on peut affirmer que la loi de Moore est tout aussi vérifiée que les lois de la thermodynamique. En se penchant davantage sur la physique conventionnelle, on peut constater un certain nombre d'incohérences par rapport à ses lois et à un certain nombre des hypothèses qui la sous-tendent. Il ne s'agit en aucune façon de déprécier la science, mais simplement de souligner ce fait qu'en science, tout, y compris les lois, est en évolution.

Quant si la loi de Moore va cesser d'exister, on peut dire d'une manière générale que les lois ne cessent pas de s'appliquer : les scientifiques les recyclent pour leur donner une portée plus générale. Il se pourrait que la loi de Moore connaisse la même transformation. Les lois qui ont un caractère linéaire ou trop simpliste s'appliquent rarement de façon générale, sachant qu'il n'existe pas de ligne droite dans la nature, ni même dans ses modèles temporels. Le plus probable est donc que la loi de Moore évoluera pour ressembler davantage à une fonction sigmoïdale et devenir mieux adaptée à la réalité.

- » **La météorologie :** Songeons à ce que peut représenter une tentative de prédire le temps qu'il fera à court terme, compte tenu du vaste nombre de mesures nécessaires, comme la température, la pression atmosphérique, l'hygrométrie, les vents et les précipitations à différents moments, les lieux et les altitudes. Les prévisions météo constituent véritablement un des premiers problèmes en matière de grandes données, et un exemple très pertinent.

D'après Weather Analytics, une compagnie qui fournit des données sur le climat, le produit intérieur brut (PIB) mondial dépend pour plus de 33 % de l'impact des conditions météorologiques sur l'agriculture, la pêche, le tourisme et les transports, pour ne citer que l'essentiel. Si l'on remonte aux années cinquante, les premiers superordinateurs servaient à traiter autant de données que possible, sachant que dans le domaine de la météorologie, plus il y a de données et plus les prévisions sont fiables. C'est la raison pour laquelle tout le monde fait provision de capacités de stockage et de traitement, comme on peut le lire à propos de l'Association météorologique coréenne, sur la page <https://www.wired.com/insights/2013/02/how-big-data-can-boost-weather-forecasting/>, concernant les prévisions météorologiques et l'étude du changement climatique.

- » **La physique :** Il n'est que de voir les quantités considérables de données produites par les expérimentations dans lesquelles on utilise les accélérateurs de particules pour tenter de déterminer la structure de la matière, de l'espace et du temps. Ainsi, par exemple, le Large Hadron Collider (<https://home.cern/topics/large-hadron-collider>), le plus grand accélérateur de particules jamais construit, produit 15 Po (petaoctets) de données par an par suite des collisions de particules (<http://home.web.cern.ch/about/computing>).
- » **La génomique :** Le séquençage d'un seul brin d'ADN, qui consiste à déterminer l'ordre précis des diverses combinaisons des quatre bases – adénine, guanine, cytosine et thymine – constituant la structure de la molécule, exige une grande quantité de données. Ainsi, un seul chromosome, une structure contenant l'ADN dans la cellule, peut nécessiter entre 50 Mo et 300 Mo. L'être humain possède 46 chromosomes, et les données relatives à l'ADN d'un seul individu utilisent tout un DVD. Imaginons simplement la capacité de stockage faramineuse que

peuvent nécessiter les données relatives à un certain nombre de personnes, ou celle qui pourrait être nécessaire pour séquencer d'autres formes de vie sur Terre (<https://www.wired.com/2013/10/big-data-biology/>).

- » **L'océanographie** : De nombreux capteurs ont été disposés dans les océans pour mesurer la température et les courants et même, avec l'aide des hydrophones, pour enregistrer les sons à des fins scientifiques (en vue de mieux connaître les poissons, les cétacés et le plancton) et militaires (détecter les sous-marins furtifs). À propos de ces efforts de surveillance, qui deviennent de plus en plus complexes et qui prennent une forme numérique, vous pouvez lire l'article suivant : <http://www.theatlantic.com/technology/archive/2014/08/listen-in-the-navy-is-tracking-ocean-sounds-collected-by-scientists/378630/>.
- » **Les satellites** : Les prises de vue depuis le ciel pour surveiller toute la surface du globe ainsi que l'atmosphère ne sont pas quelque chose de nouveau (TIROS 1, le premier satellite à avoir renvoyé des images et des données, date de 1960). Cependant, au fil des ans, le monde a lancé plus de 1 400 satellites actifs destinés à observer la Terre. La quantité de données envoyée par ces satellites est stupéfiante, et ces données ont des applications aussi bien militaires (surveillance) que civiles : suivi du développement économique, surveillance de l'agriculture, contrôle des changements et des risques. Un seul satellite de l'Agence spatiale européenne, Sentinel 1A, produit 5 Po de données en deux ans d'activité (lire sur la page <https://spaceflightnow.com/2016/04/28/europessentinel-satellites-generating-huge-big-data-archive/>).

À tout cela s'ajoutent aujourd'hui les quantités considérables de données produites ou véhiculées par l'Internet, ce qui engendre le besoin de nouvelles solutions en termes de stockage de données et d'algorithmes de traitement :

- » D'après la National Security Agency (NSA), la quantité d'informations circulant chaque jour à travers l'Internet dans le monde entier atteignait 1 826 Po de données en 2013, dont 1,6 % sous forme d'e-mails et d'appels téléphoniques. Pour assurer la sécurité des États-Unis, la NSA doit vérifier le contenu d'au moins 0,025 % de cet ensemble de courriers électroniques et d'appels téléphoniques (en recherchant les mots-clés qui pourraient révéler, par exemple, un projet terroriste). Cela représente tout de même 25 Po par an, l'équivalent de 37 500 CD-ROM de données stockées et analysées (et ce nombre croît). Pour plus de détails à ce sujet, consultez la page http://www.businessstandard.com/article/news-ani/nsa-claims-analysts-look-at-only-0-00004-of-world-s-internet-traffic-for-surveillance-113081100075_1.html.
- » L'Internet des objets est en train de devenir une réalité. Au cours des quinze dernières années, vous avez dû entendre plusieurs fois cette expression, mais la croissance du nombre de machines connectées à l'Internet est maintenant au bord de l'explosion. L'idée est de placer des capteurs et des émetteurs partout et d'exploiter les données de manière à mieux contrôler tout ce qui se passe dans le monde et à rendre les objets plus intelligents. Les dispositifs de transmission sont de plus en plus petits, de moins en moins coûteux et de moins en moins gourmands en énergie : certains sont déjà assez petits pour pouvoir être placés n'importe où (voir, par exemple, cette radio de la taille d'une fourmi que des ingénieurs de Stanford ont mise au point : <http://news.stanford.edu/news/2014/september/ant-radio-arbabian-090914.html>.) Selon les spécialistes, il y aura sur Terre en 2020 six fois plus d'objets connectés que d'individus, mais les sociétés de recherche et les panels d'experts révisent déjà ces chiffres à la hausse (<http://www.gartner.com/newsroom/id/3165317>).

Utiliser des algorithmes pour exploiter les grandes données

Le genre humain se trouve aujourd'hui confronté à un volume de données sans précédent, produit par un matériel informatique de plus en plus miniaturisé et de plus en plus puissant et analysé par des algorithmes que ce même processus a permis de développer. Ce n'est pas simplement un problème de volume, même si le volume représente déjà en lui-même un problème difficile. Tel qu'il a été formalisé par la société de recherche Gartner en 2001 puis repris et étendu par d'autres compagnies comme IBM, le concept de grandes données peut se résumer par les quatre V qui représentent leurs principales caractéristiques (pour plus de détails, consultez la page <http://www.ibmbigdatahub.com/infographic/four-vs-big-data>) :

- » **Volume** : La quantité de données.
- » **Vitesse** : La vitesse à laquelle les données sont produites.
- » **Variété** : Le nombre de sources de données et leurs différents types.
- » **Véracité** : La qualité et l'exactitude des données (quantification des erreurs, des mauvaises données, du bruit mélangé aux signaux), une mesure de l'incertitude relative aux données.



Chacune de ces caractéristiques des grandes données représente un défi et une opportunité. Le volume, par exemple, a trait à la quantité de données utiles. Ce qu'une organisation considère comme des grandes données peut représenter des petites données pour une autre organisation. Ce n'est pas parce qu'il est impossible de traiter les données sur une certaine machine que ce sont des grandes données. Les grandes données se distinguent des données habituelles en ce qu'elles obligent une organisation à réviser ses méthodes et ses solutions et sollicitent le progrès des technologies et des algorithmes.

Leur variété permet d'utiliser les grandes données pour faire des découvertes scientifiques sans suivre la méthode scientifique, comme l'explique cet article important et très débattu de Chris Anderson, alors rédacteur en chef de *Wired* : <https://www.wired.com/2008/06/pb-theory/>. L'auteur s'appuie sur l'exemple de Google, qui a pu se hisser au premier plan dans les secteurs de la publicité et de la traduction, non pas en recourant à des modèles ou à des théories spécifiques, mais en appliquant des algorithmes pour tirer des enseignements des données. À l'instar de ce qui se passe dans le domaine de la publicité, les données scientifiques (en physique, en biologie) peuvent contribuer à cette innovation permettant aux scientifiques d'aborder les problèmes non pas avec des hypothèses, mais en étudiant les variations observables dans les grands ensembles de données et par les algorithmes de découverte.

La véracité permet la démocratisation des données. Dans le passé, les organisations amassaient les données parce qu'elles étaient précieuses et difficiles à obtenir. De nos jours, la croissance en volume des données provenant de sources diverses est telle que les conserver n'a plus beaucoup de sens (90 % des données mondiales ont été créées au cours des deux dernières années), et il n'y a donc aucune raison d'en limiter l'accès. Les données deviennent un bien de consommation courante, si bien que de nombreux programmes de données en libre accès fleurissent dans le monde entier (les États-Unis ont une longue tradition de libre accès : les premiers programmes de données ouvertes remontent aux années soixante-dix, quand la National Oceanic and Atmospheric Administration, la NOAA, avait commencé à diffuser gratuitement au grand public ses données météorologiques). Cependant, les données étant devenues un produit de base, l'incertitude relative à ces données est devenue un problème. On ne sait plus quel crédit accorder aux données, dont on ne connaît parfois même pas la source.



Les données sont aujourd'hui si omniprésentes que leur valeur ne réside plus dans la réalité de l'information (comme pour les données stockées dans la base de données d'une entreprise). La

valeur qu'elles représentent est liée à l'utilisation qui en est faite. Ici, tout dépend des algorithmes. Une compagnie comme Google utilise des données librement accessibles comme le contenu des sites Web ou le texte des publications et des ouvrages. Et cependant, la valeur que Google tire de ces données provient principalement de ses algorithmes. À titre d'exemple, la valeur des données réside dans l'algorithme PageRank (voir [Chapitre 11](#)), qui est au fondement même de l'activité de Google. Cette valeur des algorithmes se vérifie pour d'autres entreprises de la même manière. Le moteur de recommandation d'Amazon contribue pour une part significative aux recettes de cette compagnie. De nombreuses sociétés financières utilisent le trading algorithmique et le robo-advice afin de mieux tirer parti des données boursières en libre accès et des informations économiques pour leurs investissements.

Gérer le flux de données

Quand les flux de données sont considérables, tout stocker peut devenir difficile, voire impossible. En réalité, ce n'est même pas utile en général. Voici quelques chiffres concernant l'activité sur Internet au cours d'une minute :

- » 150 millions d'e-mails envoyés ;
- » 350 000 nouveaux tweets envoyés sur Twitter ;
- » 2,4 millions de requêtes lancées sur Google ;
- » 700 000 personnes se sont connectées à leur compte Facebook.

Compte tenu de tels volumes, accumuler les données toute la journée en vue d'une analyse continue n'est sans doute pas la bonne méthode. Mieux vaut stocker les données quelque part pour les analyser plus tard (c'est la stratégie d'archivage typique des bases de données et des entrepôts de données). Cependant, les données recherchées sont généralement les plus récentes, et elles perdent de leur utilité avec le temps (et dans certains secteurs,

comme le secteur financier, un jour peut représenter une grande quantité de temps).

De surcroît, on peut s'attendre à recevoir demain encore plus de données (la quantité de données augmente en effet de jour en jour), si bien qu'il devient difficile, voire impossible, de trier les données et de supprimer les plus anciennes à mesure que de nouvelles données arrivent. Cette situation fait penser au châtiment de Sisyphe, dont la punition infligée par Zeus, selon la légende grecque, était de devoir éternellement pousser un immense rocher jusqu'au sommet d'une colline pour le voir ensuite redescendre à chaque fois (pour plus de détails, voir <http://www.mythweb.com/encyc/entries/sisyphus.html>).

Parfois, pour compliquer encore les choses, les données peuvent arriver si vite et en si grande quantité qu'il devient impossible de les écrire sur un disque dur. Ce problème est typique des expérimentations avec les particules menées à l'aide d'accélérateurs comme le Large Hadron Collider. Les scientifiques doivent décider quelles données conserver (<http://home.cern/about/computing/processing-what-record>).

Naturellement, il est possible de mettre les données en file d'attente pendant un certain laps de temps, mais pas trop longtemps car la file s'allongerait rapidement et deviendrait impossible à gérer. Une file d'attente stockée en mémoire, par exemple, provoquerait rapidement la saturation de la mémoire et l'affichage d'un message d'erreur.

Les flux de nouvelles données peuvent rendre obsolète le traitement des données précédentes, et la procrastination n'est pas une solution. Des stratégies variées ont été élaborées pour faire face instantanément à des quantités massives et changeantes de données :

- » **Stockage** : Une partie des données est stockée car cela peut permettre d'éclaircir certains points par la suite. Cette méthode repose sur des techniques permettant un stockage immédiat des données et une analyse très rapide par la suite, quel que soit leur volume.

- » **Synthèse** : Lorsque rien ne justifie de conserver l'intégralité des données, seules les données les plus importantes sont conservées.
- » **Consommation** : Les données qui restent sont consommées, car leur usage est prédéterminé. Les algorithmes peuvent instantanément les lire, les traiter et en tirer l'information exploitable. Ensuite, le système les oublie définitivement.

Ce livre aborde le premier point au [Chapitre 13](#), qui est consacré à la distribution des données entre divers ordinateurs et à l'étude des algorithmes utilisés à cette fin (une stratégie de type « diviser pour régner »). Les sections qui suivent abordent les deux autres points, avec une application aux données qui alimentent les systèmes informatiques.



L'arrivée massive de données dans un système informatique fait souvent l'objet d'une analogie avec l'eau : on parle de canalisation des données, de flux de données, de torrents de données, *etc.*

Le traitement de données peut d'ailleurs être comparé à la consommation de l'eau courante : quand le robinet est ouvert, on peut stocker l'eau dans des récipients ou dans des bouteilles, ou bien on peut l'utiliser pour se laver les mains, pour cuisiner, pour rincer les aliments ou pour faire la vaisselle. Dans tous les cas, la plus grande partie de l'eau s'écoule dans la vidange, et cependant cette eau aura été très utile, sinon vitale.

Analyser les flux en utilisant la bonne recette

Pour traiter les flux de données, il faut des algorithmes de flux, et ce qu'il est important de savoir concernant les algorithmes de flux est qu'en dehors de quelques calculs qu'ils effectuent avec précision, ils produisent nécessairement des résultats

approximatifs. L'output est presque correct, et si la réponse qu'il donne n'est pas la réponse exacte, elle en est néanmoins très proche.

Dans la gestion des flux, il est clair qu'il convient de se concentrer sur les mesures qui présentent un intérêt et de laisser de côté un certain nombre de détails. Certains indicateurs statistiques peuvent être intéressants, comme la moyenne, le minimum et le maximum. Par ailleurs, il peut être utile de compter les éléments dans le flux ou de distinguer les anciennes informations des nouvelles. Divers algorithmes peuvent être utilisés selon le problème à résoudre, mais la recette est toujours constituée des mêmes ingrédients :

- » **Échantillonnage** : Réduisez le flux pour que les données soient gérables ; représentez l'ensemble du flux ou les plus récentes observations à l'aide d'une fenêtre de données mobile.
- » **Hachage** : Limitez la variété des données à un ensemble fini de nombres entiers simples (voir la section « Recourir au hachage » du [Chapitre 7](#)).
- » **Synthèse** : Créez un bref résumé de l'indication dont vous avez besoin, en éliminant les détails les moins utiles. Cette méthode vous permet d'exploiter une simple mémoire de travail qui peut être la mémoire centrale de votre ordinateur ou son disque dur.

Une autre propriété des algorithmes de flux, qu'il convient de garder en mémoire, est leur simplicité. Les flux de données peuvent être très rapides. Les algorithmes qui demandent trop de calculs risquent de passer à côté de données essentielles, et ces données seront alors perdues pour toujours. En envisageant la situation sous cet angle, on peut mesurer l'utilité des fonctions de hachage sachant qu'elles transforment rapidement les inputs en quelque chose de plus facile à gérer et à rechercher, car pour ces deux opérations, le degré de complexité est $O(1)$. Les deux autres techniques relèvent du principe de *compression avec perte* (à propos de la compression, voir [Chapitre 14](#)). La compression avec

perte permet de représenter quelque chose de complexe sous une forme simplifiée. En sacrifiant un certain niveau de détail, on économise beaucoup de temps de traitement et de volume de stockage.

L'échantillonnage consiste à extraire du flux une série limitée d'exemples et à les traiter comme s'ils représentaient la totalité des données. Il s'agit d'une technique courante en statistique, qui consiste à se fonder sur un échantillon pour procéder à des inférences sur un contexte plus étendu (que l'on appelle techniquement l'*univers* ou la *population*).

Réserver les bonnes données

Les statistiques sont apparues en un temps où le recensement était impossible. Un recensement est une enquête systématique effectuée par les pouvoirs publics auprès d'une population, et qui consiste à la compter et à acquérir des données utiles la concernant : lieu de résidence, famille, vie quotidienne et activité professionnelle. Le recensement trouve son origine dans l'Antiquité. La Bible fait état d'un recensement dans le livre des Nombres, lorsque la population des Israélites fait l'objet d'un comptage après la sortie d'Égypte. À des fins de taxation, les anciens Romains effectuaient un recensement périodique de la population de leur vaste empire. Dans l'Antiquité, des documents historiques font état d'activités de recensement similaires en Égypte, en Grèce, en Inde et en Chine.

Les statistiques, et plus particulièrement les statistiques inférentielles, permettent d'obtenir le même résultat qu'un recensement avec une marge d'erreur acceptable, en interrogeant un nombre de personnes plus limité (appelé un *échantillon*). Les sondeurs peuvent ainsi déterminer l'opinion générale de la population concernée sur divers sujets, et notamment les intentions de vote à l'approche d'une élection. Aux États-Unis, par exemple, le statisticien Nate Silver a su prédire le gagnant des élections présidentielles de 2012 dans chacun des 50 États en

utilisant les données provenant d'échantillons de la population (<https://www.cnet.com/news/obamas-win-a-big-vindication-for-nate-silver-kingof-the-quants/>).

À l'évidence, un recensement représente des coûts faramineux (plus grande est la population, plus élevés sont les coûts) et suppose beaucoup d'organisation (c'est pourquoi les recensements sont rares), tandis qu'une enquête statistique auprès d'un échantillon est plus rapide et moins onéreuse. Parce qu'elles entraînent moins de coûts et d'impératifs organisationnels, les statistiques sont aussi la meilleure solution pour traiter les flux de grandes données : les utilisateurs de ces flux n'ont pas besoin de chaque bribe d'information et peuvent exploiter des données rendues moins complexes.

Le recours à des échantillons pose néanmoins un problème. L'échantillonnage, qui est un processus fondamental en statistiques, consiste à sélectionner de façon aléatoire un petit nombre d'exemples tirés de la population globale. Un principe fondamental est que chaque individu de la population doit avoir exactement la même probabilité de faire partie de l'échantillon. Supposons un échantillon d'une personne parmi une population d'un million d'habitants : pour chaque habitant, la probabilité de faire partie de l'échantillon est donc égale à un millionième. Mathématiquement, si la variable N représente la population et si n est la taille de l'échantillon, la probabilité qu'un individu soit dans l'échantillon est n/N ([Figure 12-2](#)). L'échantillon représenté est un *échantillon aléatoire simple* (à partir de ce type le plus élémentaire, on peut définir des types d'échantillons plus complexes).

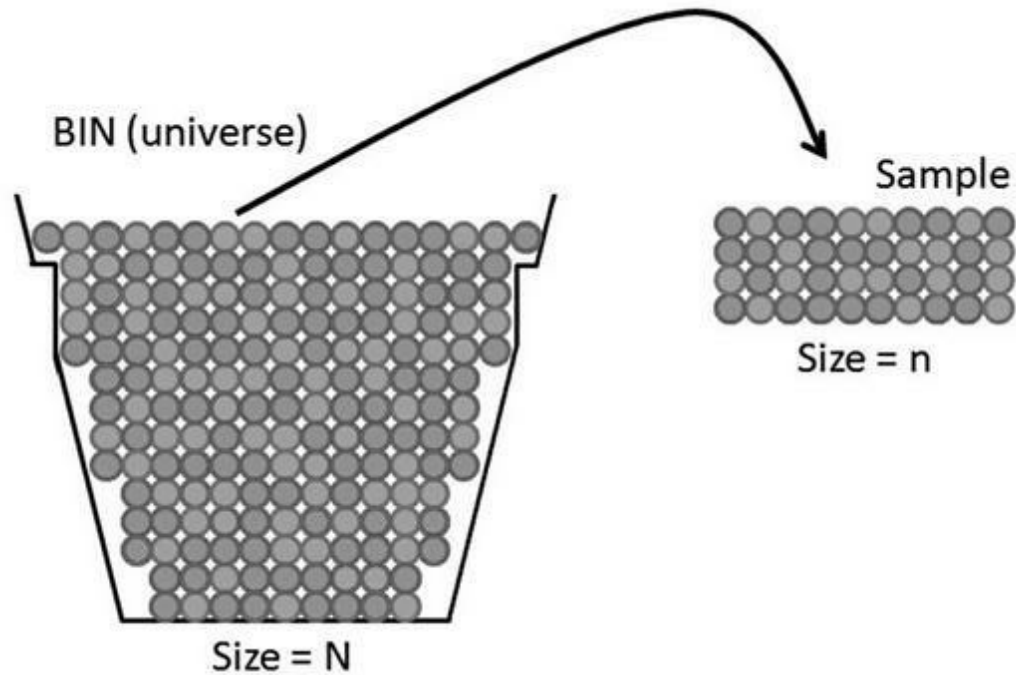


FIGURE 12-2 L'échantillonnage à partir d'une urne.

Utiliser un échantillon aléatoire simple revient à jouer à la loterie, sachant que tous les numéros doivent être dans l'urne si l'on veut pouvoir en extraire un échantillon représentatif de l'ensemble. On ne peut pas facilement faire entrer des flux de données dans un espace de stockage dont on pourra extraire un échantillon : il faut plutôt extraire l'échantillon à la volée. En réalité, il faut utiliser l'*échantillonnage à réservoir*. De la même manière qu'un réservoir retient l'eau en vue de son utilisation ultérieure et que cette eau n'est pas inerte puisque de l'eau entre dans le réservoir et de l'eau en sort, l'algorithme sélectionne des éléments au hasard pour les conserver comme échantillons jusqu'à ce que d'autres éléments arrivent pour les remplacer.

L'algorithme de l'échantillonnage à réservoir est plus élaboré que le *fenêtrage* (*windowing*), qui consiste à créer une file accueillant les nouveaux éléments ([voir Figure 12-3](#)). Les éléments plus anciens quittent la file en fonction d'un signal de déclenchement. Cette méthode s'applique quand on a besoin d'obtenir des données sur le flux à des intervalles de temps réguliers. Supposons

que vous vouliez savoir combien de pages les utilisateurs sollicitent auprès d'un serveur Internet à chaque minute. Vous allez constituer une file contenant les requêtes lancées en une minute, compter les éléments de cette file, noter le résultat, vider la file, puis la remplir à nouveau.

Une autre application du fenêtrage consiste à obtenir une quantité fixée des données les plus récentes. Dans ce cas, chaque fois qu'un élément est ajouté à la file, l'élément le plus ancien en sort. La file est une structure de type « premier entré, premier sorti » (FIFO) (voir [Chapitre 6](#)).

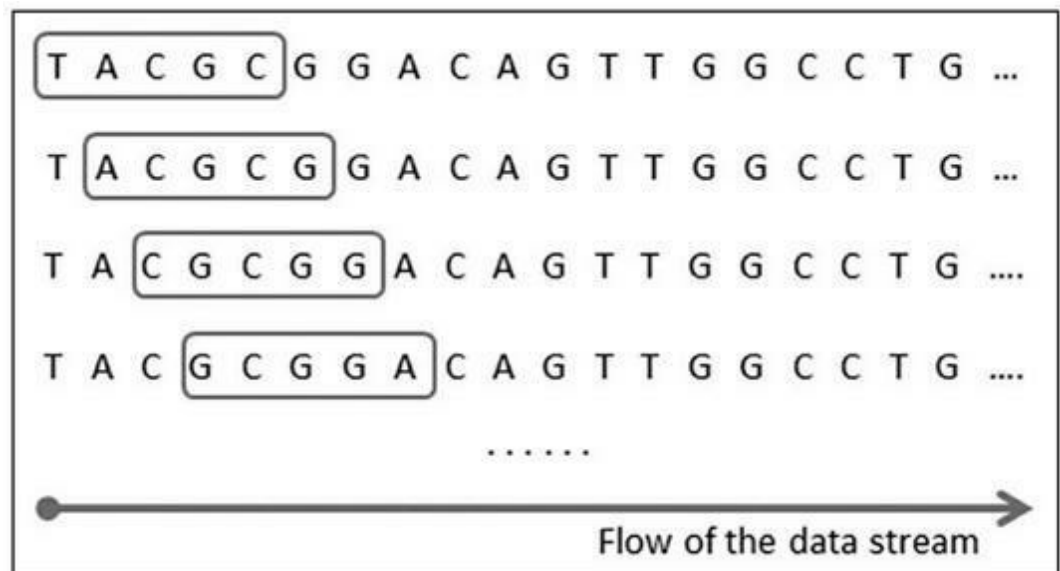


FIGURE 12-3 Un exemple de fenêtrage d'un flux de données génétiques.

Le fenêtrage consiste à traiter les échantillons à l'aide d'une fenêtre glissante : les éléments qui apparaissent dans la fenêtre représentent un certain laps de temps ou un certain segment du flux de données. L'échantillonnage à réservoir représente l'étendue totale du flux sous la forme d'une quantité de données gérable, l'échantillon statistique du flux.

Le principe de l'échantillonnage à réservoir est le suivant : on initialise l'échantillon en y mettant des éléments tirés du flux de données jusqu'à ce qu'il soit prêt. Ainsi, par exemple, si l'échantillon est constitué de 1 000 éléments, un nombre de

données qui ne sature généralement pas la mémoire interne de l'ordinateur, on commence par sélectionner les 1 000 premiers éléments du flux. Le nombre d'éléments de l'échantillon est noté k , et lorsque k éléments du flux sont sélectionnés, l'algorithme commence à exécuter les tâches suivantes :

- 1. À partir du début du flux, l'algorithme comptabilise chaque nouvel élément qui arrive. Le compte est stocké dans la variable n . La tâche suivante s'exécute quand $n = k$.**
- 2. À mesure que de nouveaux éléments arrivent, la valeur de n augmente. Pour tout nouvel élément du flux, la probabilité qu'il soit ajouté à l'échantillon réservoir est égale à k/n et la probabilité qu'il ne soit pas ajouté est égale à $(1 - k/n)$.**
- 3. La probabilité est vérifiée pour chaque nouvel élément entrant. Tout se passe comme à la loterie : soit l'élément est ajouté, soit il est écarté. S'il est ajouté, l'algorithme supprime de l'échantillon un élément plus ancien en suivant une certaine règle (la plus simple consistant à prendre un ancien élément au hasard) et le remplace par le nouvel élément.**

Le code suivant est un exemple simple d'algorithme réalisé avec Python. Il crée un échantillon de cinq éléments à partir d'une séquence de lettres de l'alphabet (en guise de flux de données) (vous retrouverez ce code dans le fichier téléchargeable A4D ; 12 ; Managing Big Data.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import string
datastream = list(string.ascii_uppercase) + list(
    string.ascii_lowercase)
print(datastream)
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
 'K', 'L',
 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
 'X',
 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
```

```
'j',  
'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',  
'v',  
'w', 'x', 'y', 'z']
```

Les chaînes mises à part, cet exemple utilise des fonctions du module aléatoire pour créer une « graine », c'est-à-dire une valeur de départ (pour des solutions stables et reproductibles) et sur la base d'un nombre entier aléatoire, il teste la nécessité de changer un élément dans le réservoir. En dehors de cette valeur de base, vous pouvez modifier la taille de l'échantillon, ou même, alimenter l'algorithme avec un flux différent (pour que l'exemple soit probant, il faut que ce soit une liste Python).

```
from random import seed, randint  
seed(9) # changer cette valeur pour obtenir des  
résultats différents  
sample_size = 5  
sample = []  
  
for index, element in enumerate(datastream):  
    # Tant que le réservoir n'est pas rempli, on ajoute  
des éléments  
    if index < sample_size:  
        sample.append(element)  
    else:  
        # Ayant rempli le réservoir, on teste un  
remplacement  
        # aléatoire en fonction des éléments  
        # observés dans le flux de données  
        drawn = randint(0, index)  
        # Si drawn est inférieur ou égal à la taille  
        # de l'échantillon, on remplace un élément  
antérieur  
        # par l'élément arrivant du flux  
        if drawn < sample_size:  
            sample[drawn] = element  
  
print (sample)  
  
['y', 'e', 'v', 'F', 'i']
```

Cette procédure vous permet de disposer à tout moment d'un bon échantillon réservoir représentatif du flux global de données. Dans

cette implémentation, la variable `index` joue le rôle de n et la variable `sample_size` joue le rôle de k . Il convient de noter deux aspects particuliers de cet algorithme :

- » Quand la variable `index` augmente, compte tenu de l'arrivée de nouvelles données, la probabilité qu'une donnée fasse partie de l'échantillon diminue. En conséquence, au début du flux, de nombreux éléments entrent et sortent de l'échantillon, mais le rythme de ce changement décroît à mesure que le flux se poursuit.
- » En recensant la probabilité d'entrée de chaque élément présent dans l'échantillon et en calculant la moyenne de toutes ces probabilités, on obtient un chiffre très proche de k/n , qui est la probabilité qu'un élément d'une population fasse partie de l'échantillon.

Obtenir une réponse des données d'un flux

L'échantillonnage est une excellente stratégie pour gérer les flux, mais cela ne permet pas de répondre à toutes les questions que peut soulever le flux de données. Ainsi, par exemple, l'échantillon ne permet pas de savoir si l'on a déjà vu un élément donné du flux, sachant qu'il ne contient pas toute l'information relative à ce flux. Il en est de même de problèmes comme le comptage du nombre d'éléments d'un flux ou du calcul de la fréquence des éléments.

Pour obtenir de tels résultats, il faut des fonctions de hachage ([Chapitre 7](#)) et des aperçus, c'est-à-dire des résumés simples et approximatifs des données. Les sections suivantes abordent le hachage. Elles vous indiquent comment savoir si un élément du flux qui se présente est déjà apparu auparavant, même si le flux est infini et s'il n'est pas possible de conserver une mémoire exacte de l'intégralité du flux antérieur.

Filtrer les éléments d'un flux par cœur

Au cœur des algorithmes de flux, on retrouve souvent les filtres de Bloom. Ils ont été inventés il y a près de 50 ans par Burton H. Bloom, en un temps où l'informatique en était encore à ses balbutiements. L'idée initiale était d'échanger de l'espace (de la mémoire) et/ou du temps (de la complexité) contre ce que Bloom a appelé des *erreurs admissibles*. Son article initial était intitulé « Space/Time Trade-offs in Hash Coding with Allowable Errors » (pour plus de détails, voir <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.2080&rank=2>).

Vous vous demandez peut-être quels sont l'espace et le temps considérés par Bloom comme justifiant son algorithme. Imaginons qu'il faille, à l'aide d'une des structures de données déjà étudiées, déterminer si un élément est déjà apparu dans un flux. Pour trouver ce que l'on cherche dans un flux, il faut une grande rapidité d'enregistrement et de recherche, c'est pourquoi la meilleure option est sans doute la table de hachage. L'utilisation d'une *table de hachage*, comme on l'a vu au [Chapitre 7](#), consiste simplement à ajouter et stocker dans un tableau les éléments que l'on veut enregistrer. L'accès à un élément dans une table de hachage est rapide, car chaque élément y est repéré par une clé, c'est-à-dire par une valeur facile à manipuler (l'élément lui-même pouvant être complexe). Stocker à la fois les éléments et un index comporte cependant des limitations. Quand une table de hachage reçoit plus d'éléments qu'elle ne peut en gérer, comme c'est le cas d'éléments d'un flux continu et potentiellement infini, des problèmes de mémoire ne tardent pas à se poser.



Une propriété essentielle des filtres de Bloom est la possibilité de faux positifs alors qu'il ne peut y avoir de faux négatifs. Supposons, par exemple, qu'un flux de données contienne des données de contrôle en temps réel d'une centrale électrique. Avec un filtre de Bloom, l'analyse du flux de données montrerait que

les valeurs attendues font probablement partie de la série des valeurs autorisées, quelques erreurs étant tolérées. Cependant, quand une erreur se produit dans le système, cette même analyse indique que les valeurs ne font pas partie des valeurs autorisées. Les faux positifs ne devraient pas poser de problème, mais l'absence de faux négatifs est un gage de sécurité. Compte tenu de la possibilité de faux positifs, les filtres comme celui de Bloom sont des structures de données probabilistes : ils fournissent non pas une réponse certaine, mais une réponse probable.



Les *hachages*, c'est-à-dire les différentes entrées d'une table de hachage, sont rapides à traiter car ils jouent le même rôle que l'index d'un livre. On utilise une fonction de hachage : l'input est un élément contenant des données complexes, et l'output est un simple nombre qui indexe cet élément. La fonction de hachage est déterministe, car elle produit le même nombre chaque fois qu'elle reçoit un input particulier. Le hachage permet de localiser l'information complexe dont on a besoin. Les filtres de Bloom sont un système simple et utile pour enregistrer les traces de nombreux éléments sans être obligé de les stocker comme le fait une table de hachage. Leurs principaux ingrédients sont les suivants :

- » **Un vecteur de bits** : Une liste d'éléments constitués de bits qui peuvent chacun prendre comme valeur 0 ou 1. La liste est constituée d'un grand nombre de bits, noté m . Ce nombre doit être le plus grand possible, mais il est possible d'en définir une taille optimale.
- » **Une série de fonctions de hachage** : Chacune de ces fonctions correspond à une valeur différente. Ces fonctions peuvent rapidement compresser les données et produire des résultats uniformément distribués entre les valeurs d'output minimum et maximum du hachage.

Ajouter des éléments aux filtres de Bloom

De façon générale, on crée des filtres de Bloom d'une taille fixe (les versions récemment mises au point permettent de redimensionner le filtre). L'utilisation d'un filtre consiste à y ajouter de nouveaux éléments et à lire les éléments qui y sont déjà présents. Il n'est pas possible de supprimer un élément du filtre après l'avoir ajouté (la mémoire du filtre est ineffaçable). Quand un élément est ajouté à un vecteur de bits, certains bits de ce vecteur prennent la valeur 1. Dans l'exemple de la [Figure 12-4](#), le filtre de Bloom ajoute X au vecteur de bits.

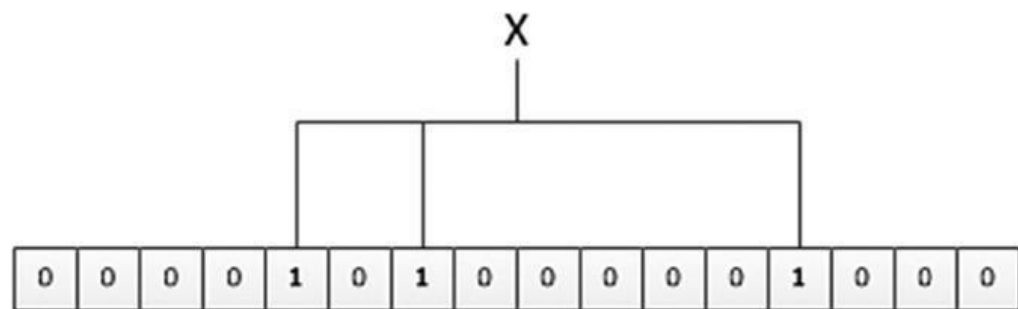


FIGURE 12-4 Ajout d'un élément à un vecteur de bits.

On peut ajouter au vecteur de bits autant d'éléments que nécessaire. La [Figure 12-5](#), par exemple, montre ce qui se produit quand on ajoute au vecteur de bits un autre élément Y. Il convient de noter que le bit 7 est le même pour X et pour Y. En conséquence, le bit 7 représente une collision entre X et Y. Ces collisions peuvent être source de faux positifs : elles font que l'algorithme peut considérer qu'un élément a déjà été ajouté au vecteur de bits alors que ce n'est pas le cas. L'utilisation d'un vecteur de bits de plus grande dimension réduit les risques de collisions et rend le filtre de Bloom plus performant, mais cela se produit au détriment de l'espace et du temps.

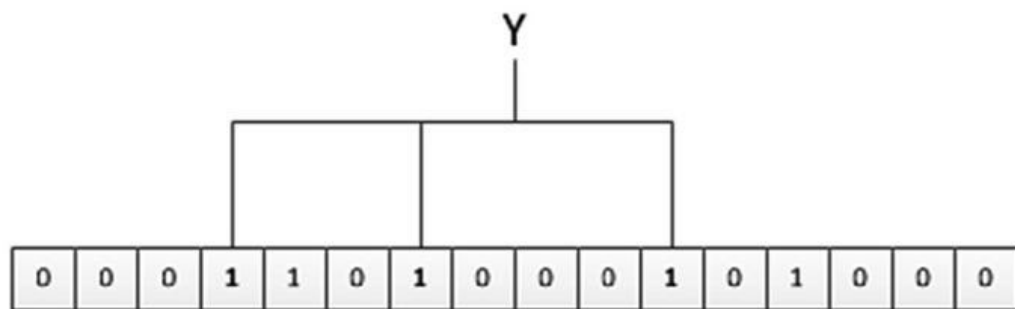


FIGURE 12-5 L'ajout d'un second élément peut engendrer des collisions.

Rechercher un élément dans un filtre de Bloom

L'examen du filtre de Bloom permet de déterminer si un élément particulier apparaît dans le vecteur de bits. Au cours du processus de recherche, l'algorithme recherche la présence d'un 0 dans le vecteur de bits. Dans la section précédente, par exemple, les éléments X et Y ont été ajoutés au vecteur de bits. Dans la recherche d'un élément Z, l'algorithme trouve un 0 au deuxième bit, comme le montre la [Figure 12-6](#). La présence d'un 0 signifie que Z ne fait pas partie du vecteur de bits.

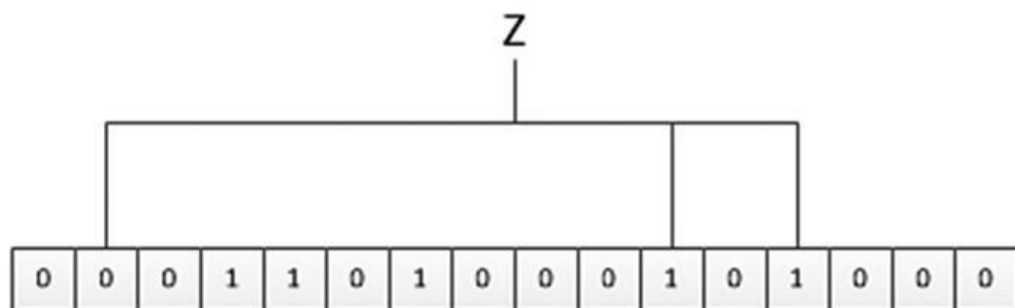


FIGURE 12-6 Localiser un élément et déterminer s'il existe revient à chercher les 0 dans le vecteur de bits.

Comment fonctionne le filtre de

Bloom ?

Cet exemple utilise Python pour illustrer le fonctionnement d'un filtre de Bloom et montre graphiquement le résultat. Prenons le cas d'un robot de recherche, c'est-à-dire d'un logiciel spécialisé qui parcourt le Web pour repérer les changements éventuels sur les sites Internet (et éventuellement recopier une partie des données de ces sites, une activité appelée le *Web scraping*). On utilise ici un vecteur de bits de dimension réduite et trois fonctions de hachage. Cette configuration n'est certes pas idéale pour gérer un grand nombre d'éléments (le vecteur de bits sera vite rempli), mais elle suffira pour cet exemple.

```
hash_functions = 3
bit_vector_length = 10
bit_vector = [0] * bit_vector_length

from hashlib import md5, sha1

def hash_f(element, i, length):
    """ C'est une fonction magique """
    h1 =
int(md5(element.encode('ascii')).hexdigest(),16)
    h2 =
int(sha1(element.encode('ascii')).hexdigest(),16)
    return (h1 + i*h2) % length

def insert_filter(website):
    result = list()
    for hash_number in range(hash_functions):
        position = hash_f(website, hash_number,
                           bit_vector_length)
        result.append(position)
        bit_vector[position] = 1
    print ('Ajouté sur les positions : %s' % result)

def check_filter(website):
    result = list()
    for hash_number in range(hash_functions):
        position = hash_f(website, hash_number,
                           bit_vector_length)
        result.append((position,bit_vector[position]))
    print ('Octets sur les positions : %s' % result)
```

Le programme commence par créer un vecteur de bits et des fonctions dont l'activité sera la suivante :

- » Générer plusieurs fonctions de hachage (en utilisant le double hachage mentionné au [Chapitre 7](#)) conformément aux algorithmes de hachage md5 et sha1.
- » Insérer un objet dans le vecteur de bits.
- » Vérifier si les octets correspondant à un objet dans le vecteur de bits sont activés.

L'ensemble de ces éléments constitue un filtre de Bloom (le vecteur de bits en étant l'élément clé). Dans cet exemple, le robot visite d'abord le site Internet wikipedia.org pour trouver des informations sur quelques pages :

```
insert_filter('wikipedia.org')  
print (bit_vector)
```

```
Ajouté sur les positions : [0, 8, 6]  
[1, 0, 0, 1, 0, 0, 1, 1, 1, 0]
```

Cette activité a pour effet d'activer les bits des positions 0, 6 et 8 du vecteur de bits. Le programme examine ensuite le site youtube.com (sur lequel ont été publiées de nouvelles vidéos de chatons) et ajoute au filtre de Bloom les informations tirées de cette visite :

```
insert_filter('youtube.com')  
print (bit_vector)
```

```
Ajouté sur les positions : [3, 0, 7]  
[1, 0, 0, 1, 0, 0, 1, 1, 1, 0]
```

Ici, le filtre de Bloom est activé sur les positions 0, 3 et 7. Compte tenu de la dimension réduite du vecteur de bits, il existe déjà une collision sur la position 0, mais les positions 3 et 7 sont totalement nouvelles. À cette étape, l'algorithme ne pouvant pas se rappeler ce qu'il a déjà visité (mais les sites visités pouvant être vérifiés à

l'aide du filtre de Bloom), le programme de cet exemple vérifie qu'il n'a pas visité yahoo.com afin d'éviter des répétitions ([Figure 12-7](#)) :

```
check_filter('yahoo.com')  
Octets sur les positions : [(7, 1), (5, 0), (3, 1)]
```

Comme l'indique la représentation graphique, on peut ici être sûr que le robot n'a jamais visité yahoo.com car le filtre de Bloom fait état d'au moins une position, la position 5, dont le bit n'a jamais été activé.



Souvent, un robot cherche à trouver un nouveau contenu sur les sites Internet et évite de copier des données qu'il a déjà enregistrées et transmises. Plutôt que de procéder à un hachage du domaine ou de l'adresse d'une seule page, on peut alimenter directement un filtre de Bloom en utilisant une partie du contenu du site, et s'en servir pour assurer le suivi des changements intervenant sur ce site par la suite.

Il existe un moyen simple et direct de faire diminuer la probabilité d'obtenir un faux positif. Il suffit d'augmenter la taille du vecteur de bits, qui constitue la partie fondamentale d'un filtre de Bloom. Davantage d'adresses signifient moins de risques de collision dans les résultats des fonctions de hachage. La dimension m du vecteur de bits peut être calculée à partir de l'estimation de n , le nombre d'objets distincts que l'on prévoit d'ajouter, m devant rester bien supérieur à n . La valeur idéale du nombre k de fonctions de hachage à utiliser pour minimiser les risques de collision peut être estimée à l'aide de la formule suivante ($\ln$ étant le logarithme népérien) :

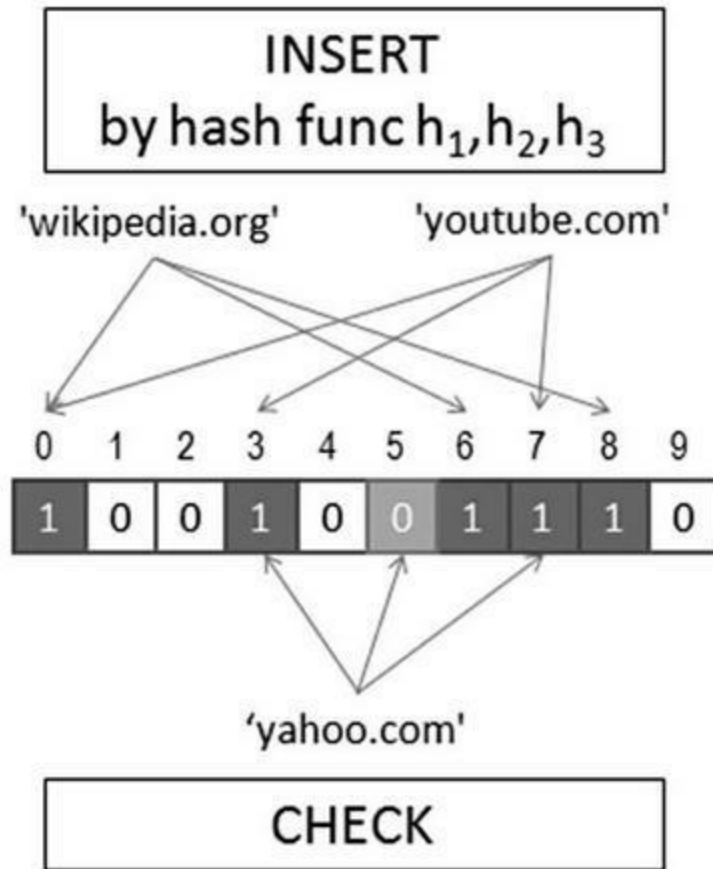


FIGURE 12-7 Tester l'appartenance à un site Web à l'aide d'un filtre de Bloom.

$$k = (m/n) * \ln(2)$$

Une fois que m , n et k ont été définis, une seconde formule permet d'estimer la probabilité d'une collision (ou fréquence de faux positifs) en utilisant un filtre de Bloom :

$$\text{Fréquence de faux positif} = (1 - \exp(-kn/m))^k$$

Si vous ne pouvez pas déterminer n en raison de la variété des données du flux, vous devez changer la valeur de m , la dimension du vecteur de bits (c'est-à-dire l'espace mémoire) ou k , le nombre de fonctions de hachage (équivalent au temps) pour corriger la fréquence de faux positifs. Ce compromis à trouver reflète les relations que Bloom étudie dans son article initial entre l'espace, le temps et le risque d'erreur.

Trouver le nombre d'éléments distincts

Même s'il permet de suivre les objets provenant d'un flux, le filtre de Bloom ne peut pas dire quel est leur nombre. Un vecteur de bits rempli de valeurs 1 peut occulter le véritable nombre d'objets hachés à la même adresse (tout dépend du nombre de hachages et du risque de collision).

Connaître le nombre d'objets est utile dans des situations variées, par exemple quand on veut savoir combien d'utilisateurs distincts ont vu une certaine page d'un site ou combien de requêtes distinctes ont été lancées sur les moteurs de recherche. Stocker tous les éléments et identifier les doublons n'est pas envisageable quand on est confronté à des millions d'éléments, surtout s'ils proviennent d'un flux. Pour connaître le nombre d'objets distincts dans un flux, il faut passer par une fonction de hachage, mais cette méthode implique une approximation numérique.

Cette approximation est ce que l'on appelle un *résumé*. Il s'agit d'obtenir en réponse une valeur inexacte, mais pas vraiment fausse. L'approximation est acceptable, car la valeur réelle n'en est pas trop éloignée. Dans cet algorithme intelligent qu'est *HyperLogLog*, un algorithme fondé sur la probabilité et l'approximation, on observe les caractéristiques des nombres issus du flux. HyperLogLog résulte des travaux de deux informaticiens, Nigel Martin et Philippe Flajolet. Flajolet a amélioré l'algorithme initial, l'algorithme de *Flajolet-Martin* (ou algorithme LogLog), pour en faire la version HyperLogLog, plus élaborée et dont le fonctionnement est le suivant :

1. **Un hachage convertit en nombre tout élément provenant du flux.**
2. **L'algorithme convertit ce nombre en nombre binaire (la norme numérique étant la base 2 pour les ordinateurs).**

3. L'algorithme compte le nombre initial de zéros dans le nombre binaire et garde la trace du nombre maximal rencontré, noté n .
4. L'algorithme estime le nombre d'éléments distincts apportés par le flux en utilisant n . Le nombre d'éléments distincts est 2^n .

Supposons que le premier élément de la chaîne soit le mot *dog*. L'algorithme le hache pour en faire une valeur entière et celle-ci est mise sous forme binaire. Le résultat est 01101010. Un seul zéro apparaît au début du nombre. L'algorithme enregistre donc 1 comme nombre maximum de zéros consécutifs rencontrés. Les mots qui suivent sont *parrot* et *wolf*, dont les équivalents binaires sont 11101011 et 01101110, si bien que n ne change pas. En revanche, quand le mot *cat* se présente, l'output est 00101110, si bien que n est maintenant égal à 2. Pour estimer le nombre d'éléments distincts, l'algorithme calcule 2^n , soit $2^2 = 4$. La [Figure 12-8](#) montre le processus.

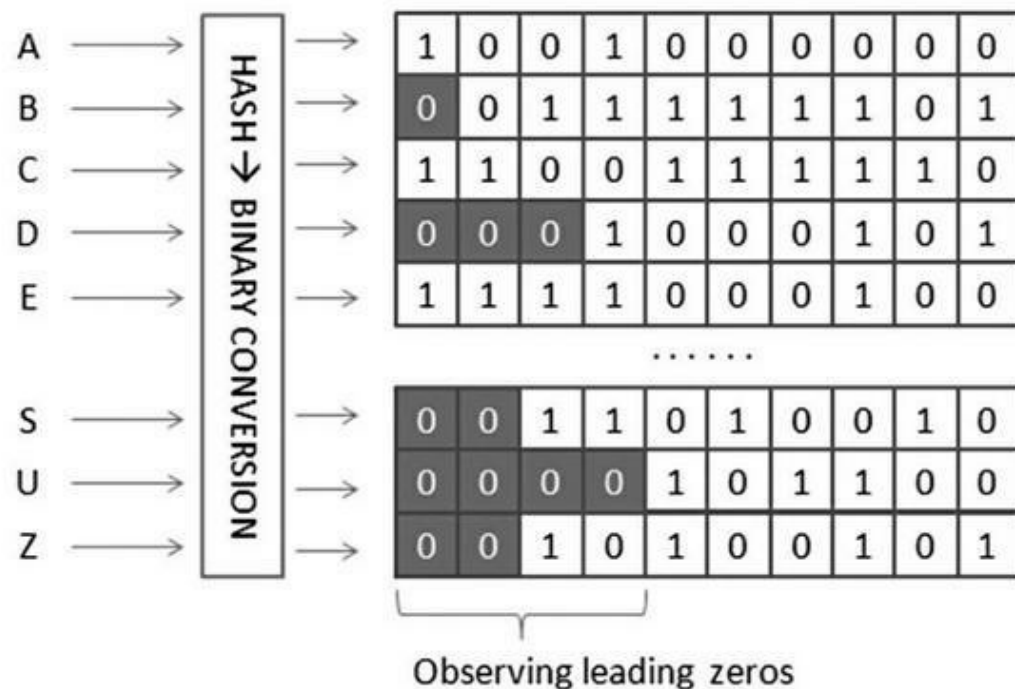


FIGURE 12-8 Compter seulement les zéros en début de chaîne.

Un avantage de cet algorithme est que si votre hachage produit des résultats aléatoires également distribués (comme dans un filtre de Bloom), l'étude de la représentation binaire permet de calculer la probabilité qu'une séquence de zéros apparaisse. La probabilité qu'un nombre binaire soit égal à 0 étant $1/2$, le calcul de la probabilité d'une séquence de zéros consiste simplement à multiplier cette probabilité de $1/2$ par elle-même autant de fois qu'il doit y avoir de zéros dans la séquence :

- » 50 % ($1/2$) de chances qu'un nombre commence par 0.
- » 25 % ($1/2 * 1/2$) de chances qu'un nombre commence par 00.
- » 12,5 % ($1/2 * 1/2 * 1/2$) de chances qu'un nombre commence par 000.
- » $(1/2)^k$ de chances qu'un nombre commence par k zéros (on utilise les puissances pour calculer rapidement les multiplications par un même nombre).



Moins il y a de nombres traités par HyperLogLog, plus grande est l'imprécision. La précision augmente quand le calcul HyperLogLog est répété un grand nombre de fois en utilisant différentes fonctions de hachage et en retenant la moyenne des réponses pour chaque calcul, mais le hachage répété prend du temps, or les flux sont rapides. Une autre possibilité consiste à utiliser le même hachage, mais en scindant le flux en deux groupes (en séparant les éléments selon leur ordre d'arrivée) et en conservant, pour chaque groupe, le nombre maximum de zéros consécutifs. À la fin, on calcule l'estimation de chaque élément pour chaque groupe, puis la moyenne arithmétique de toutes les estimations. Cette méthode est la *moyenne stochastique*. Elle donne des estimations plus précises que l'application de l'algorithme à l'ensemble du flux.

Apprendre à compter les objets dans un flux

Ce dernier algorithme du chapitre exploite aussi les fonctions de hachage et utilise aussi le principe des résumés. Au préalable, il filtre les doublons et compte les éléments distincts apparus dans le flux de données. Apprendre à compter les objets dans un flux peut vous permettre de déterminer quels sont les éléments les plus fréquents, ou de classer les événements habituels et inhabituels. Cette technique est utilisée pour résoudre des problèmes comme identifier les requêtes les plus fréquentes sur un moteur de recherche, les articles les plus vendus par un détaillant en ligne, les pages les plus appréciées d'un site Web, ou les titres boursiers les plus volatils (en comptant le nombre de ventes et d'achats d'un titre).

La solution de ce problème, *Count-Min Sketch*, est appliquée à un flux de données. Un seul passage de données est nécessaire, et le programme stocke aussi peu d'informations que possible. Cet algorithme est appliqué dans de nombreuses situations réelles (par exemple, pour analyser le trafic sur un réseau ou pour gérer des flux de données distribués). Il faut utiliser plusieurs fonctions de hachage, chacune associée à un vecteur de bits, selon une méthode analogue à celle du filtre de Bloom ([Figure 12-9](#)) :

- 1. Initialiser tous les vecteurs de bits en mettant à zéro toutes les positions.**
- 2. Appliquer la fonction de hachage pour chaque vecteur de bits à réception d'un objet provenant d'un flux. Utiliser l'adresse numérique résultante pour incrémenter la valeur à cette position.**
- 3. Appliquer la fonction de hachage à un objet et retrouver la valeur à la position associée lorsqu'il s'agit d'estimer la fréquence d'un objet. Parmi toutes les valeurs provenant d'un vecteur de bits, prendre la plus petite comme fréquence du flux.**



Des collisions étant toujours possibles quand on utilise une fonction de hachage, surtout si le vecteur de bits associé comporte peu d'emplacements, mieux vaut disposer d'un certain nombre de

vecteurs de bits afin d'être sûr qu'au moins un de ces vecteurs contient la valeur correcte. La valeur sélectionnée doit être la plus petite, car elle n'est pas mêlée à des comptages de faux positifs en raison de collisions.

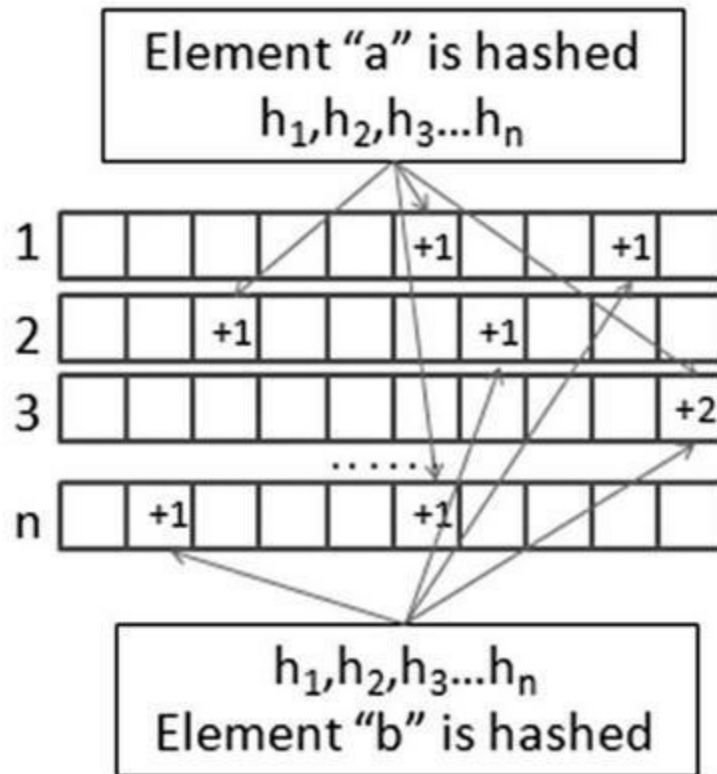


FIGURE 12-9 La mise à jour des valeurs dans un Count-Min Sketch.

Chapitre 13

Effectuer des opérations en parallèle

DANS CE CHAPITRE

- » Comprendre pourquoi ce qui est simplement plus grand, plus vaste et plus rapide n'est pas toujours la bonne solution
 - » Examiner les méthodes de stockage et de calcul des sociétés de l'Internet
 - » Trouver comment le regroupement des ressources informatiques réduit les coûts
 - » Mettre des algorithmes complexes sous forme d'opérations indépendantes pouvant être réalisées en parallèle, grâce à MapReduce
-

Pour gérer et traiter des quantités colossales de données, les méthodes de flux et d'échantillonnage présentent des avantages évidents (voir [Chapitre 12](#)). Ces algorithmes permettent d'obtenir un résultat même avec une puissance de calcul limitée (si vous utilisez votre ordinateur personnel, par exemple). Il faut cependant compter avec des coûts associés :

- » **Le traitement du flux en continu** : Il permet de gérer des quantités de données illimitées. Cependant, les algorithmes ne sont pas rapides car ils doivent traiter les éléments de données un par un, en étant tributaires de la vitesse du flux.
- » **L'échantillonnage** : Il permet d'appliquer n'importe quel algorithme sur n'importe quelle machine. Cependant, le

résultat obtenu est imprécis car obtenir la bonne réponse est probable mais pas certain. Le plus souvent, elle est seulement plausible.

Certains problèmes obligent à gérer de grandes quantités de données de manière précise et en temps opportun. Dans le monde du numérique, les exemples abondent : lancer une requête par mot-clé sur plusieurs milliards de sites Web, traiter plusieurs éléments d'information (recherche d'une image dans un enregistrement vidéo ou une correspondance entre des séquences d'ADN), *etc.* Effectuer séquentiellement de tels calculs demanderait toute une vie.

La solution consiste à recourir à l'*informatique distribuée*, c'est-à-dire à exploiter de façon simultanée les capacités de traitement d'un certain nombre d'ordinateurs interconnectés sur un réseau, et des algorithmes tournant sur ces machines de façon parallèle et indépendante.

Gérer des quantités colossales de données

L'utilisation de l'Internet pour exécuter un vaste ensemble de tâches et la popularité croissante de ses meilleures applications, notamment les moteurs de recherche et les réseaux sociaux, ont amené les professionnels de nombreux secteurs d'activité à repenser leur façon d'appliquer les algorithmes et les solutions logicielles, en vue de pouvoir faire face à une avalanche de données. Dans cette révolution, la recherche par sujet et l'établissement de liens entre les personnes jouent un rôle essentiel.

Il n'est que d'imaginer la progression, en une quinzaine d'années, du nombre de sites Internet et de pages accessibles. Même en utilisant un algorithme intelligent comme PageRank (étudié au [Chapitre 11](#)), il reste difficile de faire face à des séries de données

toujours plus vastes et plus changeantes. Il en est de même des services de réseautage proposés par des compagnies comme Facebook, Twitter, Pinterest, LinkedIn, *etc.* À mesure que le nombre d'utilisateurs s'accroît et que leurs liens réciproques se développent, le graphe sous-jacent qui les relie devient gigantesque. Sur une grande échelle, la gestion des sommets et des liens en vue d'identifier des groupes et des connexions devient incroyablement difficile (la Troisième partie de ce livre étudie les graphes en détail).

Outre les données relatives aux communications, il faut compter avec les détaillants en ligne qui proposent dans leurs entrepôts virtuels des milliers et des milliers de produits et de services (livres, films, jeux, *etc.*). L'article que vous achetez a pour vous une signification précise, mais pour le détaillant qui vous le vend, le contenu de votre panier est avant tout un problème de décision d'achat à résoudre grâce à la connaissance de vos préférences, qui lui permet de vous suggérer et de vous vendre d'autres produits à la place ou en complément.

Comprendre le paradigme du parallélisme

Pour intégrer davantage de puissance de traitement dans les microprocesseurs, les fabricants de puces électroniques ont trouvé une solution simple (anticipée et décrite en partie par la loi de Moore, qui est présentée au [Chapitre 12](#)). Cependant, faire plus gros, plus grand et plus rapide n'est pas toujours la bonne solution. Constatant que l'absorption d'énergie et le dégagement de chaleur limitaient les possibilités d'adjonction de processeurs supplémentaires autour d'une puce électronique, les ingénieurs ont trouvé un compromis, les *processeurs multicœurs*. Le principe est de créer un processeur formé de deux ou plusieurs processeurs empilés. Cette technologie a permis une vaste diffusion de l'informatique parallèle.

L'informatique parallèle existe depuis longtemps, mais elle a d'abord été réservée aux ordinateurs ultra-performants comme les supercalculateurs Cray mis au point par Seymour Cray chez Control Data Corporation (CDC) à partir des années soixante. Pour simplifier, on peut dire que deux propriétés mathématiques, l'associativité et la commutativité, reflètent bien le principe essentiel du parallélisme en informatique. Dans une addition, par exemple, on peut grouper une partie d'une somme de plusieurs nombres, ou bien changer l'ordre des éléments additionnés :

Associativité

$$2 + (3 + 4) = (2 + 3) + 4$$

Commutativité

$$2 + 3 + 4 = 4 + 3 + 2$$

Les mêmes concepts s'appliquent aux algorithmes utilisés en informatique, qu'il s'agisse d'une série d'opérations ou d'une fonction mathématique. Le plus souvent, on peut réduire l'algorithme à une forme simplifiée en appliquant ces deux propriétés, l'associativité et la commutativité ([Figure 13-1](#)). Ensuite, on peut diviser les parties, de telle sorte que des unités différentes exécutent séparément des opérations élémentaires, la somme étant effectuée à la fin.

Dans cet exemple, deux processeurs divisent une fonction simple à trois entrées (x, y et z) en exploitant à la fois l'associativité et la commutativité. L'équation solution implique le partage de données communes (CPU1 a besoin des valeurs x et y ; CPU2 a besoin des valeurs y et z), mais le traitement s'exécute en parallèle jusqu'à ce que les deux unités délivrent leurs résultats, qui sont additionnés pour obtenir la réponse.

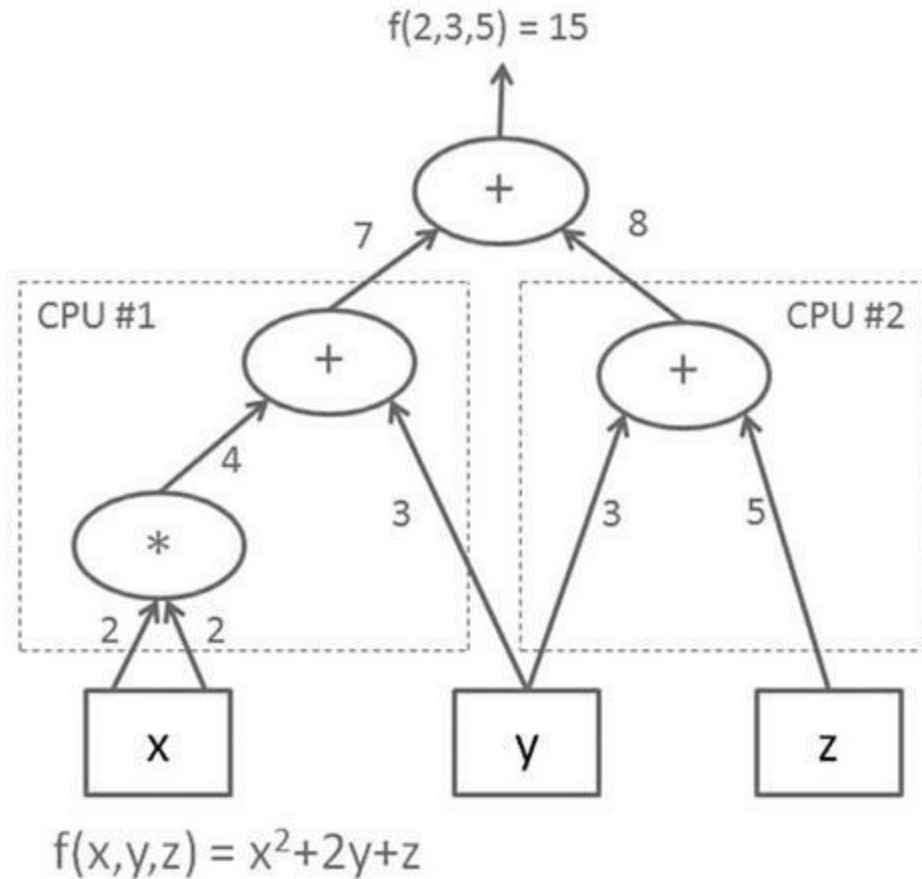


FIGURE 13-1 L'associativité et la commutativité permettent le traitement parallèle.

La logique parallèle permet le traitement simultané d'opérations de calcul en grand nombre. Plus il y a de processeurs, plus l'exécution des calculs est rapide, mais le temps nécessaire n'est pas proportionnel au nombre de processeurs parallèles (un double processeur n'exécute pas les tâches deux fois plus vite, ni un triple processeur trois fois plus vite, *etc.*). En réalité, l'associativité et la commutativité ne se vérifient pas dans toutes les parties de l'algorithme ni dans toutes les instructions du programme informatique. Simplement, tout ne peut pas être traité en parallèle, comme l'énonce la loi d'Amdahl. Cette loi permet de déterminer le gain de rapidité que représente le parallélisme (pour plus de détails, voir <http://home.wlu.edu/~whaley/classes/parallel/topics/amdahl.html>)

Par ailleurs, d'autres aspects peuvent atténuer l'effet positif du parallélisme :

- » **Surcoût** : Les résultats ne peuvent pas être additionnés en parallèle.
- » **Tâches internes** : La traduction sous-jacente d'un langage lisible par l'être humain en langage machine prend du temps. Quand deux processeurs travaillent ensemble, les coûts de cette conversion sont plus élevés, si bien qu'il est impossible d'observer un doublement de la vitesse d'exécution, même si chaque partie de la tâche peut être effectuée en parallèle.
- » **Outputs asynchrones** : Des processeurs parallèles n'exécutant pas les tâches à la même vitesse exactement, la vitesse globale de traitement est fatalement la moins rapide des deux (de la même manière que la vitesse d'une flotte est la vitesse du bateau le moins rapide).

Même s'il n'est pas toujours aussi avantageux qu'on aurait pu l'espérer, le parallélisme peut effectivement permettre de gérer un nombre colossal d'opérations plus rapidement qu'en utilisant un processeur unique (lorsqu'un grand nombre d'unités de traitement peuvent les exécuter en parallèle). Néanmoins, le parallélisme ne permet pas de traiter des quantités considérables de données sans qu'une autre solution lui soit associée : l'informatique distribuée, sur des systèmes distribués.



Quand vous achetez un nouvel ordinateur, le vendeur vous parle généralement de processeurs, de bus et de système multitâche. Le processeur est généralement un multiprocesseur, c'est-à-dire une puce constituée de deux ou plusieurs processeurs, ou « cœurs », qui travaillent en parallèle. Comme ils sont indépendants, ils exécutent les tâches de façon simultanée. Les bus font plutôt référence à la capacité d'une unité de calcul à partager son activité entre plusieurs processus, de façon quasiment parallèle. Cependant, dans ce cas, le processeur traite les canaux l'un après l'autre, si bien que les tâches ne sont pas exécutées de façon

simultanée.

Distribuer les fichiers et les opérations

Les graphes de grande dimension, les quantités colossales de fichiers texte, les images et les vidéos, et les immenses matrices d'adjacence militent pour une approche parallèle. Heureusement, il n'est plus nécessaire de disposer d'un supercalculateur, il suffit d'utiliser un ensemble d'ordinateurs bien moins puissants, mais pouvant fonctionner en parallèle. Parce que ces vastes sources de données ne cessent de croître, la solution ne réside pas dans l'utilisation d'un ordinateur unique spécialement adapté, mais dans une approche différente. Le volume des données augmente si rapidement que si l'on concevait un supercalculateur capable de les traiter, il serait déjà obsolète le jour de sa mise en service.

La solution commence par le recours à un service en ligne comme Google, Microsoft Azure ou Amazon Web Services (AWS). Pour résoudre le problème, la première étape consiste à décider où les données doivent être stockées. La deuxième étape consiste à trouver le moyen de traiter efficacement les données sans devoir trop les manipuler (sachant que le transfert d'un volume important de données d'une machine à une autre à travers l'Internet ou un réseau demande beaucoup de temps).

Ces services fonctionnent généralement de façon similaire. Les ingénieurs rassemblent un certain nombre d'idées et de concepts technologiques déjà exploités et mettent au point un système de fichiers répartis (*Distributed File System*, ou DFS). Avec un DFS, les données ne sont pas stockées sur le disque dur géant d'une machine très puissante, elles sont réparties sur un certain nombre d'ordinateurs plus petits, similaires à des ordinateurs individuels. Les ingénieurs regroupent ces ordinateurs pour former une *grappe*, un système physique d'armoires (*racks*) et de câblages. Les armoires, constituées d'ordinateurs regroupés, sont l'armature

du réseau. Une armoire comporte entre 8 et 64 ordinateurs qui sont connectés les uns aux autres. Chaque armoire est connectée aux autres par un réseau de câbles et par l'intermédiaire de plusieurs strates de commutateurs (*switches*), qui sont des dispositifs capables de gérer les échanges de données entre les armoires ([Figure 13-2](#)).

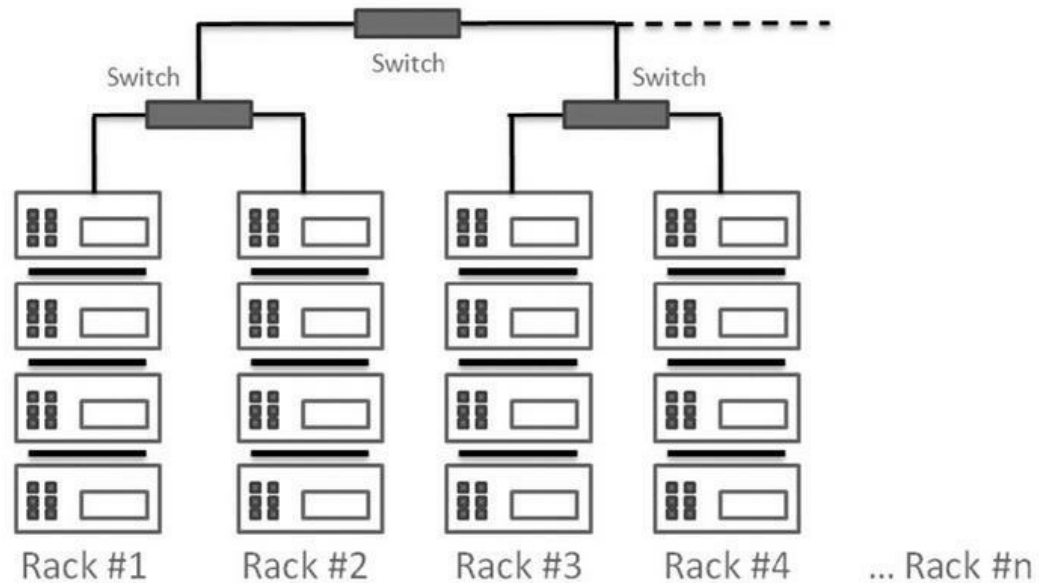


FIGURE 13-2 Un schéma représentant une grappe informatique.

Tout ce matériel, qui permet de faire fonctionner l'infrastructure de DFS, est en vente dans les magasins de matériel informatique. Théoriquement, on doit pouvoir trouver plus d'un million d'ordinateurs interconnectés sur un réseau (à propos de la version Google de cette configuration, voir <http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers/>). Il est intéressant de remarquer que ces services permettent de disposer de davantage de puissance de traitement en cas de besoin, non pas par la création de nouveaux réseaux, mais par l'addition d'ordinateurs supplémentaires.

Dans ce système, à mesure que les données arrivent, le DFS les divise en paquets (pouvant chacun atteindre la taille de 64 Mo).

Le DFS duplique ces paquets et distribue les copies à différentes unités du réseau. Le partage, la duplication et la distribution des données sont très rapides, quelle que soit la façon dont ces données sont structurées (qu'elles soient bien ordonnées, classées ou mélangées). La seule condition pour que le traitement soit efficace est que les adresses des paquets soient enregistrées par le DFS, grâce à un index créé pour chaque fichier (lui-même dupliqué et distribué) et qu'on appelle le nœud maître. La vitesse d'exécution du DFS est liée à la manière dont le DFS gère les données. Contrairement aux techniques plus anciennes de stockage de données (comme les entrepôts de données), le DFS ne nécessite aucun tri, aucun ordonnancement, aucun nettoyage des données :

- » Il gère les données quelle que soit leur dimension, car elles sont divisées en paquets faciles à gérer.
- » Il stocke les nouvelles données en les empilant sur les données plus anciennes, sans jamais devoir mettre à jour ces dernières.
- » Il duplique les données sans se soucier des redondances, si bien qu'il n'est pas nécessaire d'effectuer des sauvegardes : la duplication elle-même constitue une sauvegarde.



Un ordinateur peut être sujet à divers types de pannes : défaillance du disque dur, du processeur, du système d'alimentation, de tel ou tel composant. Statistiquement, un serveur fonctionne correctement pendant 1 000 jours en moyenne (soit environ trois ans). Par conséquent, dans un service constitué d'un million d'ordinateurs, on peut s'attendre à ce que chaque jour, un millier d'ordinateurs tombent en panne. C'est pourquoi le DFS distribue plusieurs copies de vos données à un certain nombre d'ordinateurs du réseau. La duplication réduit le risque de perte de données en cas de défaillance. La probabilité que les ordinateurs qui stockent le même paquet de données tombent tous en panne est voisine de un sur un milliard (dans l'hypothèse où le DFS copie trois fois les données), un risque assez infime pour être acceptable.

Opter pour la solution MapReduce

Même si les systèmes distribués stockent rapidement l'information, retrouver les données est un processus bien plus lent, surtout lorsqu'il est nécessaire de procéder à une analyse et d'appliquer des algorithmes. Le même type de problème surgit quand on éparpille les éléments d'un puzzle (ce qui est très facile). Il faut ensuite trier et sélectionner les pièces pour reconstruire l'image originale (ce qui est difficile et long). La gestion des données avec un DFS s'effectue comme suit :

- 1. Accéder au nœud maître et le lire pour déterminer la localisation des parties du fichier.**
- 2. Distribuer une instruction de recherche aux ordinateurs du réseau afin d'obtenir les paquets de données préalablement stockés.**
- 3. Rassembler au niveau d'un même ordinateur les paquets de données stockés sur plusieurs machines différentes (dans la mesure du possible, sachant que parfois, certains fichiers peuvent être trop volumineux pour être stockés sur une seule machine).**

Naturellement, ce processus peut devenir complexe, c'est pourquoi les ingénieurs du Web ont conclu qu'il n'était pas souhaitable de reconstituer les fichiers avant de les traiter. Une solution plus subtile consiste à les laisser sous forme de paquets sur l'ordinateur source pour qu'ils soient traités tels quels. Seule une version reduce, déjà presque entièrement traitée, sera transférée à travers le réseau, afin de limiter la transmission de données. MapReduce est la solution pour utiliser des algorithmes en parallèle sur un système de données réparties. MapReduce est un algorithme constitué de deux parties, map et reduce.

Map, qu'est-ce que c'est ?

La première phase de l'algorithme MapReduce est la partie map, une fonction que l'on retrouve dans divers *langages de programmation fonctionnels* (un style de programmation utilisant une logique de fonction mathématique). La partie map est simple : elle est basée sur un tableau unidimensionnel (dans Python, ce peut être une liste) et une fonction. En appliquant cette fonction à chaque élément du tableau, on obtient un tableau de forme identique dont les valeurs sont changées. L'exemple suivant est constitué d'une liste de dix nombres que la fonction transforme en puissances :

```
L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
m = list(map(lambda x: x**2, L))
print(m)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

La fonction map applique la fonction Python lambda (une fonction *lambda* étant une fonction définie à la volée) pour transformer chaque élément de la liste initiale. La [Figure 13-3](#) montre le résultat de ce processus de cartographie.

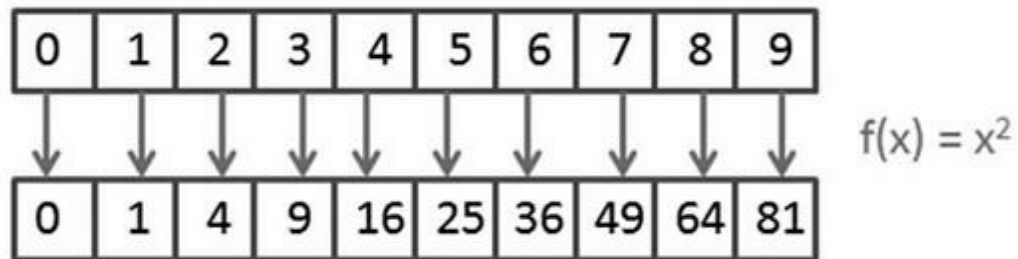


FIGURE 13-3 Cartographie d'une liste de nombres par une fonction carré.

UNE SOLUTION GLOBALE POUR MAPREDUCE

Même si ce livre montre comment aboutir *ex nihilo* à une solution MapReduce, il n'est pas nécessaire de réinventer la roue chaque fois qu'il s'agit d'exécuter cette tâche. Des modules comme MrJob (<https://pythonhosted.org/mrjob/>) permettent d'exécuter les tâches de MapReduce rapidement et facilement. En outre, vous pouvez faciliter l'exécution de la tâche à l'aide de ressources dans le nuage comme Amazon Web Services en utilisant le module Elastic MapReduce (EMR) (<https://aws.amazon.com/emr/>) ou en utilisant Hadoop (<http://hadoop.apache.org/>). Il importe de comprendre comment fonctionne l'algorithme, et c'est bien le sujet de ce livre, mais il n'est peut-être pas nécessaire d'écrire tout le code.

Il convient de noter que chaque transformation d'un élément de la liste est indépendante des autres. La fonction peut être appliquée aux éléments dans n'importe quel ordre (le résultat doit cependant être stocké dans la bonne position, dans le tableau final). Le fait que les éléments de la liste puissent être traités dans un ordre quelconque permet d'envisager naturellement une logique parallèle sans effort particulier.



Tous les problèmes ne se prêtent pas à un traitement en parallèle, et pour certains problèmes ce ne sera jamais le cas. Néanmoins, il est parfois possible de repenser ou de redéfinir le problème afin d'aboutir à une série de calculs que l'informatique pourra traiter de façon parallèle.

Et reduce, qu'est-ce que c'est ?

La seconde phase de l'algorithme MapReduce est la partie reduce (il existe aussi une étape intermédiaire, mélange et tri, qui sera expliquée plus loin mais qui n'a pas d'importance pour l'instant). À partir d'une liste, reduce applique une fonction selon une séquence de production cumulative des résultats. La fonction reduce applique donc une fonction de somme à tous les éléments de la liste. Elle prend les deux premiers éléments du tableau et les combine, puis elle combine ce résultat partiel avec l'élément

suivant du tableau, et ainsi de suite jusqu'à ce que le tableau soit complété.



On peut aussi fournir en entrée un nombre de départ. Dans ce cas, reduce commence par combiner ce nombre avec le premier élément de la liste pour obtenir le premier résultat partiel. L'exemple suivant utilise le résultat de la phase de cartographie et le réduit à l'aide d'une fonction de somme ([Figure 13-4](#)) :

0	1	4	9	16	25	36	49	64	81
---	---	---	---	----	----	----	----	----	----

$$f(x,y) = x+y$$

0 → 5 → 14 → 30 → 55 → 91 → 140 → 204 → 285

FIGURE 13-4 Réduire une liste de nombres à sa somme.

```
from functools import reduce
L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
m = list(map(lambda x: x**2, L))
r = reduce(lambda x, y: x+y, m)
print(r)
```

285



La fonction reduce traite le tableau d'input comme s'il s'agissait d'un flux de données (voir [Chapitre 12](#)). De façon générale, elle traite un élément à la fois et consigne les résultats intermédiaires.

Des opérations de hachage, de tri et de réduction

Entre les phases map et reduce, il existe une phase intermédiaire qui consiste à mélanger et trier les éléments. Dès qu'une tâche de cartographie est terminée, le système hôte redirige les tuples de paires clés/valeurs résultants vers le bon ordinateur du réseau pour la phase reduce. Cela se fait généralement en groupant les paires

correspondantes sous forme d'une liste unique et en appliquant à la clé une fonction de hachage, d'une manière similaire aux filtres de Bloom (voir [Chapitre 12](#)). L'output est une adresse dans la grappe pour le transfert des listes.

À l'autre extrémité de la transmission, l'ordinateur qui exécute la phase reduce reçoit les listes de tuples d'une ou plusieurs clés. Il y a plusieurs clés quand le hachage engendre une collision, ce qui se produit lorsque des clés différentes donnent la même valeur de hachage et se retrouvent donc sur le même ordinateur. L'ordinateur chargé de la phase reduce les trie sous forme de listes contenant la même clé avant de les traiter ([Figure 13-5](#)).

Comme le montre ce schéma, MapReduce utilise des inputs multiples au niveau de chaque ordinateur de la grappe informatique qui les stocke, cartographie les données et les transforme en tuples de paires clés/valeurs. Le système hôte transmet ces tuples, groupés sous forme de listes, à d'autres ordinateurs du réseau, lesquels exécutent des opérations de tri et de réduction pour aboutir au résultat.

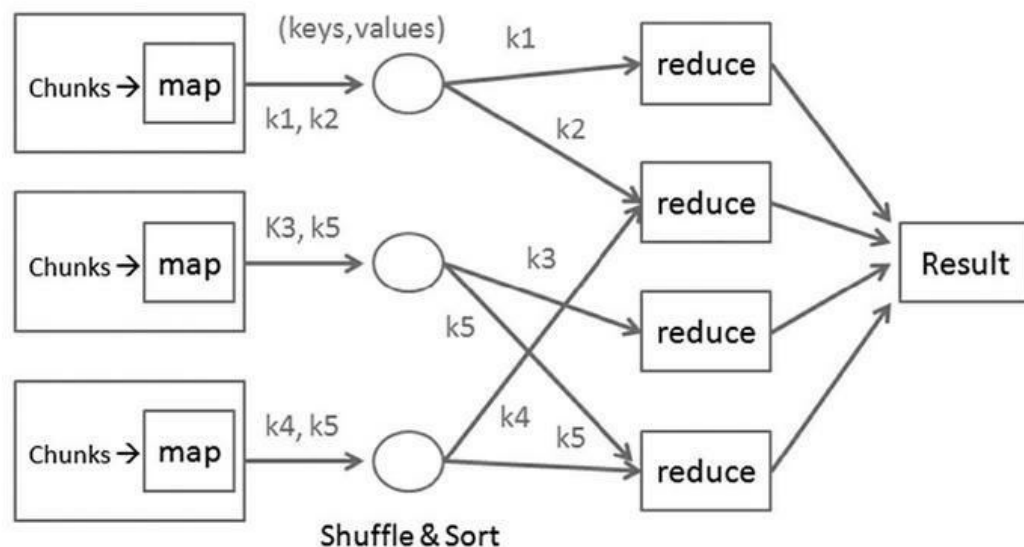


FIGURE 13-5 Un aperçu du traitement complet par la méthode MapReduce.

Étudier des algorithmes pour MapReduce

Contrairement à d'autres exemples qui figurent dans ce livre, on peut envisager MapReduce comme un type de traitement informatique ou comme un cadre de grandes données plutôt que comme un algorithme. En tant que cadre de données, MapReduce vous permet de combiner différents algorithmes distribués (des algorithmes parallèles qui répartissent les tâches de calcul entre différents ordinateurs) de telle sorte qu'ils traitent efficacement de grandes quantités de données. Les algorithmes de MapReduce se retrouvent dans un certain nombre d'applications, et pour en savoir davantage, vous pouvez consulter une page consacrée à Hadoop, avec une présentation de la société qui l'utilise, de la façon dont il est utilisé et du type de grappe informatique sur laquelle il est utilisé : <http://wiki.apache.org/hadoop/PoweredBy>. Les possibilités sont certes nombreuses, mais MapReduce est le plus souvent utilisé pour exécuter les tâches suivantes :

- » Algorithmes du texte, pour diviser les textes en éléments (jetons), créer des index, et rechercher des mots ou des expressions pertinentes.
- » Création de graphes et algorithmes graphiques.
- » Exploration de données et apprentissage de nouveaux algorithmes à partir des données (apprentissage machine).

L'algorithme MapReduce est souvent utilisé pour traiter du texte. L'exemple proposé dans cette section montre comment venir à bout d'une tâche simple, compter certains mots dans un extrait de texte à l'aide de l'approche « map » et « reduce » et exploiter l'informatique multitâche ou multiprocesseur (selon le système d'exploitation installé sur votre ordinateur).



Le langage de programmation Python n'est pas le langage informatique idéal pour les traitements en parallèle. Techniquement, en raison de problèmes de synchronisation et de

partage de l'accès à la mémoire, l'interpréteur de Python n'est pas *thread safe*, ce qui signifie qu'il n'est pas à l'abri d'erreurs lors de l'exécution d'applications utilisant des processus multiples ou des fils d'exécution sur des unités de traitement multiples. Par conséquent, Python limite le processus à un seul fil d'activité, ou *thread* (le code est distribué mais sans qu'il y ait augmentation des performances) et le parallélisme multicœur par processus multiples est vraiment difficile à obtenir, surtout avec des ordinateurs tournant sous Windows. Pour en savoir plus sur les fils d'exécution, lisez l'article de Microsoft sur la page [https://msdn.microsoft.com/fr-fr/library/windows/desktop/ms681917\(v=vs.85\).aspx](https://msdn.microsoft.com/fr-fr/library/windows/desktop/ms681917(v=vs.85).aspx).

Mettre en place une simulation de MapReduce

Cet exemple traite un texte du domaine public provenant du site de Project Gutenberg, une association à but non lucratif (<https://www.gutenberg.org/>). Le premier texte traité est le roman *Guerre et paix* de Léon Tolstoï (Leo Tolstoy en anglais, Lev Tolstoj en russe). Le code suivant charge les données en mémoire :

```
import urllib.request
url =
'http://gutenberg.readingroo.ms/2/6/0/2600/2600.txt'
response = urllib.request.urlopen(url)
data = response.read()
text = data.decode('utf-8')[627:]

print (text[:37])

WAR AND PEACE

By Leo Tolstoy/Tolstoi
```

Soyez patient ! Il faut du temps pour charger ce livre (essayez donc de le lire aussi rapidement que le fait l'ordinateur). Une fois

le chargement terminé, le code affiche les premières lignes ainsi que le titre. Le programme stocke les données dans la variable `text`. Le processus, entre autres tâches, divise le texte en mots et les stocke dans une liste :

```
words = text.split()
print ('Nombre de mots : %i' % len(words))

Nombre de mots : 566218
```

La variable des mots contient maintenant des mots du livre. Il s'agit ensuite d'importer les modules Python et les fonctions nécessaires pour cet exemple, à l'aide du code suivant :

```
import os
if os.name == "nt":
    #Safer multithreading on Windows
    from multiprocessing.dummy import Pool
else:
    #Multiprocessing on Linux,Mac
    from multiprocessing import Pool

from multiprocessing import cpu_count
from functools import partial
```

Selon votre système d'exploitation, cet exemple fonctionnera dans un contexte multiprocesseur ou multitâche. Windows utilise le *multithreading*, qui consiste à diviser la tâche en plusieurs fils d'activité (*threads*) qui seront traités simultanément par la même unité de traitement. Sous Linux et sur les systèmes Mac, le code exécute les tâches en parallèle, et chaque opération est prise en charge par une unité de traitement (cœur) différente.

Le code qui suit compte les mots d'une liste correspondant à une série de mots-clés. Après avoir supprimé toute la ponctuation, le programme compare les mots. S'il trouve une correspondance avec un mot-clé, la fonction retourne un tuple constitué d'une clé, du mot correspondant et d'une valeur unitaire, qui est un comptage. Cet output représente le cœur de la cartographie MapReduce :

```

def remove_punctuation(text):
    return ''.join([l for l in text if l not in ['.',
        ', ', '!', '?', '"']])

def count_words(list_of_words, keywords):
    results = list()
    for word in list_of_words:
        for keyword in keywords:
            if keyword == remove_punctuation(
                word.upper()):
                results.append((keyword,1))
    return results

```

Les fonctions qui suivent partitionnent les données. Cette méthode est similaire au partitionnement des données par un système distribué. Le programme distribue les calculs et rassemble les résultats :

```

def Partition(data, size):
    return [data[x:x+size] for x in range(0, len(data),
        size)]
def Distribute(function, data, cores):
    pool = Pool(cores)
    results = pool.map(function, data)
    pool.close()
    return results

```

Enfin, les fonctions suivantes mélangent et trient les données pour synthétiser les résultats. Cette étape est constituée des deux phases d'une procédure MapReduce :

```

def Shuffle_Sort(L):
    # Shuffle
    Mapping = dict()
    for sublist in L:
        for key_pair in sublist:
            key, value = key_pair
            if key in Mapping:
                Mapping[key].append(key_pair)
            else:
                Mapping[key] = [key_pair]
    return [Mapping[key] for key in Mapping]
def Reduce(Mapping):
    return (Mapping[0][0], sum([value for (key, value
        ) in Mapping]))

```

Le recours à la cartographie

Le code suivant simule un système distribué utilisant plusieurs unités de traitement. Il consiste à demander d'abord au système d'exploitation le nombre d'unités disponibles. Ce nombre peut varier. Les ordinateurs actuels en comportent généralement quatre ou huit.

```
n = cpu_count()
print (' Vous disposez de %i unités de traitement pour
MapReduce' % n)
```

Vous disposez de 4 unités de traitement pour MapReduce



Pour des raisons techniques, si vous exécutez ce code sous Windows, une seule unité sera sollicitée et vous ne profiterez pas du nombre total d'unités de traitement. La simulation semblera fonctionner, mais vous ne constaterez aucune augmentation de la vitesse de traitement.

Pour débiter le traitement, le code définit d'abord l'opération de cartographie. Il distribue ensuite la fonction de cartographie aux différents threads, et chacun d'eux traite une partition des données initiales (la liste contenant les mots du livre *Guerre et paix*). Le programme trouve les mots *peace* (paix), *war* (guerre) (dans *Guerre et paix*, y a-t-il plus de guerre que de paix, ou l'inverse ?), *Napoleon* (Napoléon) et *Russia* (Russie) :

```
Map = partial(count_words,
               keywords=['WAR', 'PEACE', 'RUSSIA',
                        'NAPOLEON'])
map_result = Distribute(Map,
                        Partition(
                            words, len(words)//n+1), n)
print ('map_result est une liste constituée de %i
éléments' %
      len(map_result))
print ('Visualisation d'un élément : %s'%
      map_result[0][:5])
```

```
Map est une liste constituée de 4 éléments
Visualisation d'un élément : [('WAR', 1), ('PEACE',
1), ('WAR', 1),
('WAR', 1), ('RUSSIA', 1)]]
```

Au bout d'un moment, le programme affiche les résultats. En l'occurrence, la liste affichée contient quatre éléments car le système hôte est constitué de quatre unités (vous pourrez voir apparaître moins d'éléments ou plus d'éléments, selon le nombre d'unités de traitement de votre machine). Chaque élément de la liste est lui-même une liste contenant les résultats de la cartographie de cette partie des mots. L'aperçu d'une de ces listes montre qu'il s'agit d'une séquence de clés (en fonction du mot-clé trouvé) et de valeurs unitaires. Les clés ne sont pas classées, elles apparaissent dans l'ordre dans lequel le code les a générées. Par conséquent, avant de soumettre les listes à la phase *reduce* pour additionner les résultats, le programme met les clés dans l'ordre et les envoie à l'unité de traitement appropriée :

```
Shuffled = Shuffle_Sort(map_result)
print ('Shuffled est une liste constituée de %i
éléments' %
      len(Shuffled))
print ('Visualisation du premier élément : %s'%
      Shuffled[0][:5])
print ('Visualisation du second élément : %s'%
      Shuffled[1][:5])
Shuffled est une liste constituée de 4 éléments
Visualisation du premier élément : [('RUSSIA', 1),
('RUSSIA', 1),
('RUSSIA', 1), ('RUSSIA', 1), ('RUSSIA', 1)]]
Visualisation du second élément : [('NAPOLEON', 1),
('NAPOLEON',
1), ('NAPOLEON', 1), ('NAPOLEON', 1), ('NAPOLEON',
1)]]
```

Comme on le voit dans cet exemple, la fonction `Shuffle_Sort` crée une liste constituée elle-même de quatre listes qui contiennent chacune les tuples qui constituent un des quatre mots-clés. Dans le contexte d'une grappe, ce traitement équivaut à faire passer chaque nœud par les résultats émis, et en utilisant un adressage (à l'aide d'une fonction de hachage par exemple, comme on l'a vu

au [Chapitre 12](#) pour le vecteur de bits d'un filtre de Bloom), à envoyer (phase du mélange) les données du tuple au nœud de réduction approprié. Le nœud récepteur place chaque clé dans la liste appropriée (phase de tri) :

```
result = Distribute(Reduce, Shuffled, n)
print ('Les résultats émis sont : %s' % result)
```

```
Les résultats émis sont : [('RUSSIA', 156),
 ('NAPOLEON', 469),
 ('WAR', 288), ('PEACE', 111)]
```

La phase de réduction additionne les tuples répartis et triés et renvoie la somme totale pour chaque clé, comme le montre le résultat affiché par le code qui reproduit une fonction MapReduce. On peut constater que dans *Guerre et paix*, Tolstoï parle de guerre plus souvent que de paix, mais mentionne Napoléon plus souvent encore.

Vous pouvez facilement reproduire cette expérience sur d'autres textes, ou même hacher la fonction de cartographie afin d'appliquer au texte une fonction différente. Vous pourriez, par exemple, analyser les romans de Sir Arthur Conan Doyle pour tenter de découvrir combien de fois Sherlock Holmes y utilise l'expression « Elementary, Watson » (« Élémentaire, mon cher Watson ») :

```
import urllib.request
url = "http://gutenberg.pglafr.org/1/6/6/1661/1661.txt"
text = urllib.request.urlopen(url).read().decode(
    'utf-8')[723:]
words = text.split()

print (text[:65])
print ('\nNombre total de mots : %i' % len(words))
Map = partial(count_words,
    keywords=['WATSON', 'ELEMENTARY'])
result = Distribute(Reduce,
    Shuffle_Sort(Distribute(Map,
        Partition(words, len(words)//n), n)),
    1)
print ('Résultats émis : %s' % result)
```

THE ADVENTURES OF SHERLOCK HOLMES
by
SIR ARTHUR CONAN DOYLE

Nombre total de mots : 107431

Résultats émis : [('WATSON', 81), ('ELEMENTARY', 1)]

Le résultat réserve des surprises ! En réalité, on ne trouve jamais cette expression dans les romans. C'est une accroche qui a été ajoutée par la suite dans les scénarios des films :
<http://www.phrases.org.uk/meanings/elementary-my-dear-watson.html>.

Chapitre 14

Compresser les données

DANS CE CHAPITRE

- » Comprendre comment les ordinateurs peuvent stocker l'information en économisant de l'espace
 - » Créer un code efficace et intelligent
 - » Exploiter les statistiques et construire des arbres de Huffman
 - » Compresser et décompresser à la volée grâce à l'algorithme de Lempel-Ziv-Welch (LZW)
-

La dernière décennie a été marquée par une avalanche de données au niveau mondial. Finalement, les données sont aujourd'hui ce qu'a été le pétrole, et des spécialistes de toutes sortes espèrent en tirer de nouvelles connaissances et de nouvelles richesses. Il s'ensuit que partout, des données sont amassées, et souvent archivées dès qu'elles arrivent. Ce stockage de données, parfois inapproprié, est lié à la capacité accrue de stocker l'information : il est devenu possible de se procurer pour un prix très modique des supports de grande capacité permettant de tout stocker, ce qui est utile comme ce qui ne l'est pas. Les Chapitres [12](#) et [13](#) traitent des déterminants de cette avalanche de données, expliquent comment gérer des flux massifs de données et présentent les méthodes utilisées pour répartir les données sur des grappes d'ordinateurs connectés ainsi que des techniques qui vous permettent de traiter les données de façon rapide et efficace.

Cependant, les données n'ont pas toujours été aussi facilement accessibles. Au cours des décennies précédentes, le stockage de données exigeait d'importants investissements dans des dispositifs de stockage de masse qui étaient coûteux (disques durs, bandes magnétiques, disquettes, disques compacts), mais dont les capacités étaient très limitées. Le stockage de données exigeait aussi une certaine efficacité (économiser de l'espace disque, c'était économiser de l'argent), et les algorithmes de compression de données étaient une solution pour pouvoir en stocker davantage sur un support, au prix du temps de traitement de l'ordinateur. Échanger de l'espace disque contre du temps permettait de réduire les coûts.

Les algorithmes de compression ont longtemps été un sujet de débat et ils sont maintenant considérés comme une solution classique dans le domaine de l'informatique. Même si les disques de stockage ont aujourd'hui des capacités plus grandes et coûtent moins cher, ces algorithmes jouent toujours un rôle non négligeable dans la transmission mobile de données et trouvent une utilisation partout où existent des goulots d'étranglement ou des coûts de mémoire élevés. La compression est pratique également lorsque le développement des infrastructures ne suit pas la croissance du volume des données, ce qui se produit en particulier concernant les réseaux mobiles et sans fil dans les pays en développement. En outre, la compression permet de transmettre plus rapidement des pages Web, de télécharger plus facilement les vidéos, de stocker des données sur un appareil mobile et de réduire les coûts des transmissions de données sur les réseaux de téléphonie mobile. Ce chapitre vous explique comment fonctionne la compression de données et à quel moment vous en avez besoin.

Rendre les données moins volumineuses

Les données informatisées sont constituées de bits, c'est-à-dire de séquences de zéros et de un. Ce chapitre explique de manière plus détaillée que les chapitres précédents l'utilisation des zéros et des un pour créer des données, sachant que la compression tire parti de ces zéros et de ces un de multiples façons. Pour comprendre le principe de la compression, vous devez savoir de quelle manière l'ordinateur crée et stocke des nombres binaires. Les sections qui suivent traitent de l'utilisation des nombres binaires en informatique.

Comprendre le codage

Dans le système binaire, les zéros et les un sont les seuls chiffres. Ils représentent les deux états possibles dans un circuit électrique : absence et présence d'électricité. Les ordinateurs ont été conçus à partir de circuits simples constitués de tubes ou de transistors : l'utilisation du système binaire à la place de notre système décimal a facilité les choses. Nous pouvons utiliser nos dix doigts pour compter de 0 à 9, et quand nous devons continuer nous ajoutons une unité à gauche. Peut-être n'y aviez-vous jamais songé, mais il est possible de compter en utilisant les puissances de dix. Le nombre 199, par exemple, peut s'exprimer sous la forme suivante :

$$10^2 * 1 + 10^1 * 9 + 10^0 * 9 = 199$$

En effet, on peut séparer les centaines des dizaines et des unités en multipliant chaque chiffre par la puissance de dix relative à sa position : 10^0 pour les unités, 10^1 pour les dizaines, 10^2 pour les centaines, *etc.*



Sachant cela, vous comprenez mieux les nombres binaires car leur logique est exactement la même, si ce n'est qu'ils utilisent non pas les puissances de dix, mais les puissances de deux. Le nombre 11000111, par exemple, est simplement

$$\begin{aligned} 2^7 * 1 + 2^6 * 1 + 2^5 * 0 + 2^4 * 0 + 2^3 * 0 + 2^2 * 1 + 2^1 * 1 + 2^0 * 1 &= \\ 128 * 1 + 64 * 1 + 32 * 0 + 16 * 0 + 8 * 0 + 4 * 1 + 2 * 1 + 1 * 1 &= \end{aligned}$$

$$128+64+4+2+1 = 199$$

Tout nombre peut être représenté sous forme binaire pour l'ordinateur. Une valeur binaire occupe l'espace mémoire requis par sa longueur totale. Le nombre 199, par exemple, s'écrit sur 8 chiffres en binaire, chaque chiffre étant un bit et 8 bits constituant un octet. La machine ne connaît les données que sous forme de bits car ses circuits ne connaissent que deux états. Cependant, à un autre niveau, le logiciel peut interpréter les bits comme des lettres, des idéogrammes, des images, des films ou des sons, et c'est là qu'intervient le codage.

Le *codage* utilise une séquence de bits pour représenter autre chose que le nombre exprimé par cette séquence. On peut représenter une lettre, par exemple, en utilisant une séquence particulière de bits. Selon la norme ASCII (*American Standard Code for Information Interchange*), la lettre A est généralement représentée par le nombre 65, soit 01000001 en binaire. On peut voir des séquences utilisées par le système ASCII à l'adresse <http://www.asciitable.com/>. Le code ASCII utilise 7 bits pour le codage (et 8 bits, soit un octet, pour la version étendue), ce qui signifie qu'on peut représenter 128 caractères différents (256 avec la version étendue). Python peut représenter la chaîne de caractères « Hello World » sous forme d'une série d'octets :

```
print (''.join(['{0:08b}'.format(ord(l))  
               for l in "Hello World"]))
```

```
0100100001100101011011000110110001101111001000000101011  
101111011100100110110001100100
```

Quand on utilise l'ASCII étendu, l'ordinateur sait qu'une séquence de 8 bits représente un caractère. Il peut donc diviser la séquence en octets de 8 bits et convertir ces octets en caractères à l'aide d'un tableau de conversion appelé *table de symboles*.



Le code ASCII permet de représenter l'alphabet occidental, mais il ne supporte pas les caractères accentués ni la richesse de certains alphabets non européens comme les idéogrammes utilisés

en chinois et en japonais. Vous utilisez probablement un système de codage élaboré comme UTF-8 ou une autre forme de codage de type Unicode (pour plus de détails, voir <http://unicode.org/>). Le codage Unicode est le système de codage par défaut de Python 3.

Avec un système de codage complexe, vous devez utiliser des séquences plus longues qu'avec le code ASCII. Selon le codage que vous choisirez, la définition d'un caractère pourra nécessiter jusqu'à 4 octets (32 bits). Pour représenter de l'information textuelle, l'ordinateur crée de longues séquences de bits. Il décode facilement chaque lettre sachant que le codage utilise des séquences de longueur fixe, qui sont stockées dans un même fichier. Certaines méthodes de codage comme Unicode Transformation Format 8 (UTF-8) peuvent utiliser des nombres de bits variables (en l'occurrence, entre 1 et 4). Pour plus de détails sur UTF-8, consultez la page <http://www.fileformat.info/info/unicode/utf8.htm>.

Étudier les effets de la compression

L'utilisation de séquences de caractères de taille fixe laisse place à d'importantes améliorations. On n'utilise pas toujours toutes les lettres de l'alphabet, et certaines lettres sont utilisées plus souvent que d'autres. Ce sont des aspects qu'exploitent les techniques de compression. En utilisant des séquences de caractères de longueur variable, on peut grandement réduire la taille d'un fichier. Cependant, un traitement complémentaire sera nécessaire pour remettre le fichier sous un format non compressé qui soit compatible avec les applications concernées. La compression supprime les espaces dans les chaînes de caractères, d'une manière organisée et méthodique, et la décompression restitue ces espaces. Lorsqu'il est possible de compresser et de décompresser les données sans qu'il n'y ait aucune perte, cela s'appelle la *compression sans perte*.

Ce même principe de compression s'applique aux images et aux sons lorsqu'il s'agit de traiter des séquences de bits d'une certaine

taille pour représenter des détails d'une vidéo ou reproduire une seconde d'un son sur les haut-parleurs de l'ordinateur. Les vidéos sont simplement des séquences de bits, chaque séquence de bits étant un *pixel*. Les pixels sont autant de petits points dont une image est constituée. De même, un message audio est constitué de séquences de bits représentant des échantillons. Un fichier audio stocke un certain nombre d'échantillons par seconde pour recréer le son. Pour plus de détails sur le stockage de vidéos et de messages audio, voir <http://kias.dyndns.org/comath/44.html>. Les ordinateurs stockent les données sous un certain nombre de formats prédéfinis, qui sont constitués de longues séquences de bits (communément appelées *flux binaires*). Les algorithmes de compression peuvent exploiter les caractéristiques de chaque format pour obtenir le même résultat en passant par un format adapté et plus compact.

Les données qui représentent des images et des sons peuvent être compressées davantage encore en supprimant des détails qui ne peuvent pas être traités. Compte tenu de nos limitations visuelles et auditives, nous risquons peu de remarquer cette perte de détails imposée par une telle compression des données. Vous savez sans doute que la compression MP3 vous permet de stocker des collections entières de CD sur votre ordinateur ou sur un appareil mobile. Le format de fichier MP3 est une simplification du format WAV initialement utilisé par les ordinateurs, qui est volumineux. Les fichiers WAV contiennent toutes les ondes sonores reçues par l'ordinateur, mais le format MP3 permet d'économiser de l'espace en supprimant et en compactant des ondes qui sont pour nous inaudibles (pour plus de détails sur le format MP3, lire l'article de la page <http://arstechnica.com/features/2007/10/theaudiofile-understanding-mp3-compression/>).



La suppression de certains détails dans les données constitue une *compression avec perte*. Les formats JPEG, DjVu, MPEG, MP3 et WMA sont autant d'algorithmes de compression avec perte spécialisés dans un type particulier de données média (images, vidéos, sons), et il en existe bien d'autres encore. La compression avec perte ne pose pas de problème pour les données destinées à

l'être humain. Toutefois, le processus n'est pas réversible. Vous pouvez obtenir une bonne compression d'une photo numérique et afficher cette photo sur votre écran de façon tout à fait satisfaisante, mais si vous l'imprimez sur une feuille de papier, vous remarquerez que la qualité de cette photo compressée, si elle reste acceptable, n'est plus la même que celle de l'original. L'output affiché sur l'écran est de 96 points par pouce (ppp), or une imprimante produit généralement un output de l'ordre de 300 à 1 200 ppp (ou davantage). Les effets d'une compression avec perte deviennent visibles avec l'impression parce que l'imprimante est capable de les faire apparaître d'une manière perceptible à nos yeux.



Le choix entre compression avec perte et sans perte est important. Supprimer les détails est une bonne méthode pour les médias, mais pas aussi probante pour les textes, car la perte de mots ou de lettres peut en altérer la signification (pour la même raison, la suppression de détails n'est pas envisageable dans le cas des langages de programmation et des instructions de programme). Si la compression avec perte est une bonne solution quand les détails ne sont pas importants, elle n'est pas envisageable lorsque la signification précise d'un message doit être préservée.

LES AVANTAGES D'UNE COMPRESSION AVEC PERTE

La compression avec perte peut être avantageuse dans le domaine de la photographie, par exemple. Un fichier image brut contient toute l'information initialement produite par le capteur d'images de votre appareil. Il n'a subi aucune forme de compression. Supposons qu'un fichier constitué d'une photo prise avec votre appareil occupe 29,8 Mo d'espace sur votre disque dur. Un fichier brut utilise souvent l'extension .raw, qui montre qu'il n'a subi aucune transformation. Si vous ouvrez ce fichier puis le sauvegardez au format .jpeg, sa taille ne sera plus que de 3,7 Mo, mais ce ne sera pas sans perte. Pour réduire la taille du fichier tout en conservant un certain niveau de détail, vous pourrez

cependant opter pour le format de fichier .jpeg en réduisant la taille du fichier à 12,4 Mo, ce qui constituera un bon compromis.

Choisir un type particulier de compression

Les algorithmes de compression sans perte compressent simplement les données pour réduire leur volume et les décompressent pour les restituer dans leur état initial. La compression sans perte trouve davantage d'applications que la compression avec perte car elle est utilisable dans tous les cas (même dans le cas d'une compression avec perte, une compression sans perte est ensuite appliquée à ce qui reste après la suppression des détails). De même qu'il existe de nombreux algorithmes de compression avec perte spécialisés dans différents médias, il existe de nombreux algorithmes de compression sans perte qui exploitent certaines caractéristiques des données (pour avoir une idée de ce qu'est la grande famille des algorithmes de compression sans perte, consultez la page http://ethw.org/History_of_Lossless_Data_Compression_Algorith



Il est essentiel de ne pas oublier que la finalité des deux types de compression, avec et sans perte, est d'éliminer la redondance dans les données. Plus il y a de redondances, plus la compression est efficace.

Il y a des chances pour que plusieurs programmes de compression de données sans perte soient installés sur votre ordinateur, ces programmes créant des fichiers en ZIP, LHA, 7-Zip ou RAR, et peut-être vous demandez-vous lequel est le meilleur. Il n'existe pas un choix qui soit meilleur que tous les autres, dans la mesure où les séquences de bits peuvent être utilisées de différentes manières pour représenter l'information. Par ailleurs, la meilleure méthode de compression ne sera pas la même selon les séquences de bits à traiter. Ce problème est évoqué au [Chapitre 1](#). Le choix le plus approprié dépendra du contenu à compresser.

Pour voir comment la compression varie selon l'échantillon fourni, essayez le même algorithme sur plusieurs échantillons de texte. L'exemple suivant, avec Python, utilise l'algorithme ZIP pour compresser le texte des *Aventures de Sherlock Holmes*, d'Arthur Conan Doyle, puis pour réduire la taille d'une séquence de lettres générée de façon aléatoire (le code source complet téléchargeable se trouve dans la section « Compression Performances » du fichier A4D ; 14 ; Compression.ipynb : pour plus de détails, consultez l'Introduction).

```
import urllib.request
import zlib
from random import randint
url = "http://gutenberg.pglafl.org/1/6/6/1661/1661.txt"
sh = urllib.request.urlopen(url).read().decode('utf-8')
sh_length = len(sh)
rnd = ''.join([chr(randint(0,126)) for k in range(sh_length)])

def zipped(text):
    return len(zlib.compress(text.encode("ascii")))

print ("Longueur initiale des deux textes : %s caractères" % sh_length)
print ("Les Aventures de Sherlock Holmes jusqu'à %s" % zipped(sh))
print ("Fichier aléatoire jusqu'à %s " % zipped(rnd))

Longueur initiale des deux textes : 594941 caractères
Les Aventures de Sherlock Holmes jusqu'à 226824
Fichier aléatoire jusqu'à 521448
```

L'output de cet exemple est instructif. Si l'application permet de réduire la taille de cette courte histoire à moins de la moitié de sa taille initiale, la réduction du texte aléatoire est bien moindre (alors que les deux textes avaient initialement la même longueur). L'output montre que l'algorithme ZIP exploite les caractéristiques du texte écrit, mais n'est pas aussi performant avec le texte aléatoire, qui n'a pas de structure prédictible.



L'efficacité de la compression de données peut être mesurée en calculant le taux de compression : il suffit de diviser la nouvelle taille du fichier par sa taille initiale. Le taux de compression indique si l'algorithme permet d'économiser beaucoup d'espace, mais un algorithme performant a aussi besoin de temps pour exécuter cette tâche. Si votre temps est précieux, sachez que la plupart des algorithmes vous permettent d'arbitrer entre le taux de compression et la rapidité de la compression et de la décompression. Dans l'exemple qui précède, portant sur le texte de Sherlock Holmes, le taux de compression est de $226824 / 594941$, soit environ 0,381. La méthode compress utilisée dans cet exemple comporte un second paramètre optionnel, `level`, qui contrôle le niveau de compression. Vous pouvez modifier la valeur de ce paramètre pour arbitrer entre la rapidité d'exécution de la tâche et le taux de compression atteint.

Bien choisir la méthode de codage

L'exemple de la section précédente montre ce qui se produit quand on applique l'algorithme ZIP à un texte aléatoire. Les résultats montrent l'efficacité de la compression. En faisant le tour des algorithmes de compression disponibles sur le marché, on peut découvrir quatre raisons à cela :

- » **Réduction du codage des caractères :** La compression fait que les caractères occupent moins de bits parce qu'elle les code selon un certain critère, comme la commodité d'utilisation. Par exemple, si vous n'utilisez qu'une partie des caractères, vous pouvez réduire le nombre de bits de manière à refléter cette utilisation partielle. On observe la même différence entre l'ASCII, qui utilise 7 bits, et l'ASCII étendu qui utilise 8 bits. Cette solution est particulièrement efficace pour des problèmes dans lesquels on peut envisager un meilleur codage que le codage standard, par exemple pour le codage de l'ADN.

- » **Réduction des longues séquences de bits identiques :** La compression utilise un code spécial pour identifier les copies multiples des mêmes bits et les remplacer par une copie unique à laquelle elle associe le nombre de répétitions. Cette option est très efficace pour les images (en particulier pour les images faxées en noir et blanc) et pour toutes les données qu'il est possible de réorganiser en regroupant les caractères similaires (c'est le cas pour les données génétiques).
- » **Exploitation des statistiques :** La compression code les caractères fréquemment utilisés sous une forme réduite. La lettre *E*, par exemple, apparaît plus souvent que toutes les autres en anglais et en français. Par conséquent, en la codant sur 3 bits seulement au lieu de 8 bits, on économise un espace considérable. C'est la méthode utilisée par le codage de Huffman, consistant à recréer la table de symboles et à économiser de l'espace, en moyenne, les caractères les plus courants occupant moins de place.
- » **Codage efficace des longues séquences de caractères qui sont fréquentes :** Cette méthode est similaire à la réduction des longues séquences de bits identiques, sauf qu'elle est utilisée avec des séquences de caractères plutôt qu'avec des caractères pris un à un. C'est la méthode utilisée par LZW, qui assimile les structures de données à la volée et crée un codage court pour de longues séquences de caractères.

Pour comprendre comment repenser le codage pour permettre d'améliorer la compression, commençons par nous intéresser à la première raison. Les scientifiques qui travaillaient sur le projet du génome humain vers 2008 (<https://www.genome.gov/10001772/all-about-the-human-genome-project-hgp/>) ont réussi à réduire considérablement le volume de leurs données en utilisant une simple astuce de codage. La cartographie de l'ensemble de l'ADN humain est ainsi devenue plus simple, ce qui a permis aux

scientifiques d'en savoir davantage sur la vie, la maladie et la mort telles qu'elles sont écrites dans nos cellules.

Les scientifiques décrivent l'ADN en utilisant des séquences formées par les lettres A, C, T et G (qui représentent les quatre nucléotides présents dans tout être vivant). Le génome humain contient six milliards de nucléotides (associés en couples qu'on appelle des bases) qui occupent un volume de plus de 50 Go quand on utilise le codage ASCII. La représentation des lettres A, C, T et G en code ASCII se fait comme suit :

```
print (' '.join(['{0:08b}'.format(ord(l))  
                 for l in "ACTG"]))
```

```
01000001 01000011 01010100 01000111
```

La somme de la ligne qui précède donne 32 bits, mais comme l'ADN est codé sur quatre caractères seulement, on peut utiliser 2 bits pour chaque caractère et économiser ainsi 75 % des bits initialement prévus :

```
00 01 10 11
```



Ce gain d'espace illustre l'intérêt de choisir le codage approprié. En l'occurrence, le codage convient bien, car l'alphabet de l'ADN est constitué de quatre lettres et il n'est donc pas justifié d'utiliser la table ASCII qui est basée sur 8 bits. Quand l'alphabet ASCII complet est nécessaire, il n'est pas possible de compresser les données en redéfinissant le codage utilisé. Il faut alors recourir au système de compression de Huffman.

Si vous ne pouvez pas réduire le codage du caractère (ou si vous l'avez déjà fait), vous pouvez toujours réduire les séquences longues. Voyez comment des données binaires permettent de répéter des séquences longues de un et de zéros :

```
00000000 00000000 01111111 11111111 10000011 11111111
```

Ici, la séquence part de zéro. On peut donc compter le nombre de zéros, puis compter les un qui les suivent, puis compter à nouveau

les zéros, et ainsi de suite. La séquence n'étant constituée que de zéros et de un, on peut la remplacer par une séquence de comptes et compresser ainsi les données. Dans cet exemple, on obtient la série de valeurs 17 15 5 10. En convertissant cette série en octets, on réduit les données initiales selon un processus facilement réversible :

```
00010001 00001111 00000101 00001010
```

Au lieu de 6 octets, il ne faut plus que 4 octets pour représenter les mêmes données. Avec cette méthode, il faut limiter le comptage maximum à 255 valeurs consécutives :

- » On peut coder chaque séquence sur un octet.
- » La première valeur est un zéro quand la séquence commence par un 1 et non par un 0.
- » Quand un bloc de valeurs contient plus de 255 éléments, on insère une valeur 0 (afin que le décodeur bascule sur l'autre valeur avec un comptage de 0 puis recommence à compter la première valeur).

Cet algorithme, appelé *Run-length encoding* (RLE), ou en français le codage par plages, est très efficace quand les données comportent un certain nombre de longues répétitions. Il a connu un grand succès dans les années quatre-vingt, car il permettait de réduire les temps de transmission des télécopies. Les télécopieurs ne transmettaient que des images en noir et blanc, et par les lignes téléphoniques terrestres. Les textes et les images étaient constitués de longues séquences de zéros et de un qu'il était pratique de réduire de cette manière. De nos jours, les entreprises n'utilisent plus que rarement la télécopie, mais les scientifiques se servent toujours de l'algorithme RLE pour compresser les données génétiques, en combinaison avec le Burrows-Wheeler Transform (un algorithme élaboré, voir <http://marknelson.us/1996/09/01/bwt/>), qui réorganise (de façon réversible) la séquence génomique sous forme de longues séries du même nucléotide. Le RLE est également utilisé pour la

compression d'autres formats de données comme JPEG et MPEG (pour plus de détails, voir <http://motorscript.com/mpeg-jpeg-compression/>).



L'efficacité d'un algorithme de compression dépend des caractéristiques des données concernées. En sachant comment fonctionnent les algorithmes et en étudiant les caractéristiques de vos données, vous pouvez choisir l'algorithme ou la combinaison d'algorithmes qui produira le meilleur résultat.

Utiliser la méthode de compression de Huffman

Redéfinir un codage, comme dans la cartographie des nucléotides de l'ADN, est un choix intelligent, mais qui n'est envisageable que si l'on n'a pas besoin d'utiliser la totalité de l'alphabet que le codage doit représenter. David A. Huffman a mis au point une autre méthode pour le codage des lettres, des chiffres et des symboles, qui est efficace même quand tous ces caractères sont utilisés. Il a réalisé cette performance en 1952 alors qu'il était étudiant au MIT, dans le cadre d'un devoir demandé par son professeur, Robert M. Fano. Robert Fano et un autre éminent scientifique, Claude Shannon (le père de la théorie de l'information), étaient confrontés au même problème.

Dans son article, intitulé *A Method for the Construction of Minimum-Redundancy Codes*, Huffman décrit en seulement trois pages sa méthode de codage époustouflante. Cette méthode a changé notre façon de stocker les données jusqu'à la fin des années quatre-vingt-dix. Les détails de cet algorithme incroyable sont présentés dans un article paru en septembre 1991 dans la revue *Scientific American* : <http://www.huffmancoding.com/my-uncle/scientific-american>. La méthode de Huffman repose sur trois grandes idées :

- » **Coder les symboles fréquents par des séquences de bits plus courtes.** Si la lettre *a* apparaît souvent dans votre

texte, par exemple, tandis que la lettre *z* y apparaît rarement, vous pouvez coder *a* sur deux bits tout en réservant un octet entier (voire plus) pour *z*. Quand les caractères fréquents sont codés sur des séquences réduites, le stockage global du texte nécessite moins d'octets qu'avec le code ASCII.

- » **Coder des séquences plus courtes à l'aide d'une unique série de bits.** Quand on utilise des séquences de bits de longueur variable, il faut s'assurer qu'une séquence réduite ne risque pas d'être interprétée de façon incorrecte, sachant que les séquences réduites sont similaires aux séquences plus longues. Ainsi, par exemple, si la lettre *a* devient 110 en binaire et si *z* devient 110110, il ne faut pas que la lettre *z* soit confondue avec une série de deux lettres *a* successives. La méthode de Huffman permet d'éviter ce problème en utilisant des codes sans préfixe : l'algorithme ne réutilise jamais les séquences réduites comme parties initiales de séquences plus longues. Si *a* est codé par 110, alors *z* sera codé par 101110 et non pas par 110110.
- » **Gérer un codage sans préfixe à l'aide d'une méthode spécifique.** Le codage de Huffman gère des codes sans préfixe grâce à l'utilisation judicieuse d'arbres binaires. L'arbre binaire est une structure de données étudiée dans les Chapitres 6 et 7. L'algorithme de Huffman utilise des arbres binaires (appelés arbres de Huffman) sous une forme élaborée. Pour plus de détails, lisez le tutoriel sur la page https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.htm



L'algorithme utilisé pour exécuter le codage de Huffman est un processus itératif qui fait référence aux *tas*, qui sont des structures arborescentes de données (évoquées au [Chapitre 6](#)). Un tas est une structure de données complexe. Compte tenu de leur utilisation possible pour organiser les données, les tas sont utiles dans le cadre d'une méthode gloutonne. Dans le chapitre suivant, consacré aux algorithmes gloutons, vous allez tester vous-même le

codage de Huffman à l'aide des exemples pratiques contenus dans le code téléchargeable qui accompagne ce livre (l'exemple de compression de Huffman se trouve dans le fichier A4D ; 15 ; Greedy Algorithms.ipynb : pour plus de détails, consultez l'Introduction).

Pour le moment, à titre d'exemple d'output du codage de Huffman, la [Figure 14-1](#) représente l'arbre binaire de Huffman utilisé pour coder une séquence longue de lettres *ABCDE* distribuées de telle sorte que *A* soit plus fréquente que *B*, *B* plus fréquente que *C*, *C* plus fréquente que *D*, et *D* plus fréquente que *E*.

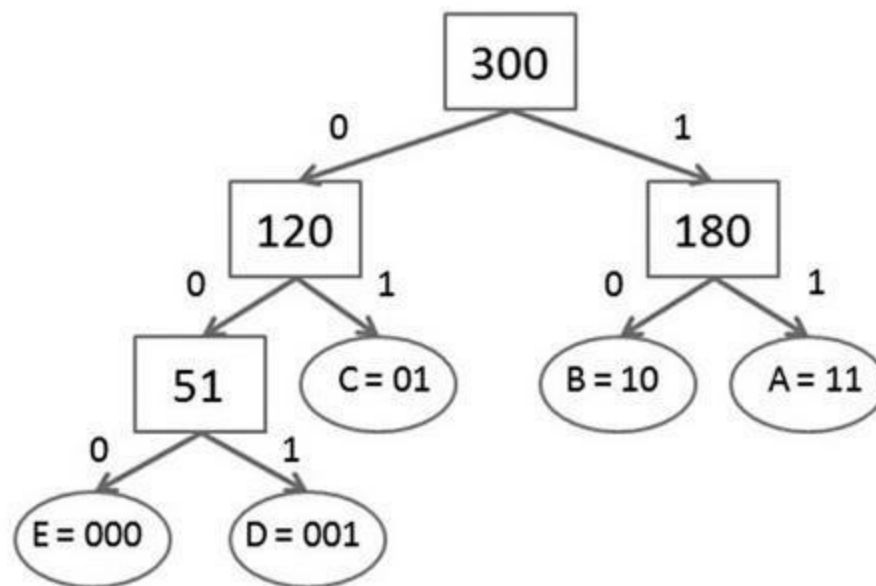


FIGURE 14-1 Un arbre de Huffman et sa table de conversion des symboles.

Les rectangles représentent les *nœuds de branches*, là où l'algorithme place le nombre de lettres restantes qu'il distribue aux *nœuds fils* (ceux qui se trouvent au-dessous des nœuds de branche dans la hiérarchie). Les ovales représentent les *nœuds feuilles*, là où se trouvent les lettres qui ont été codées avec succès. L'arbre commence à la racine, avec 300 lettres à distribuer (c'est la longueur du texte). Il distribue les lettres en dirigeant les

bits 0 le long de la branche gauche et les bits 1 le long de la branche droite jusqu'à avoir atteint toutes les feuilles nécessaires au codage. En effectuant une lecture depuis l'extrémité supérieure de la séquence des branches jusqu'à une lettre particulière, on détermine la séquence binaire qui représente cette lettre. Aux lettres les moins fréquentes (*D* et *E*) sont attribuées les séquences binaires les plus longues.



En parcourant l'arbre de Huffman de bas en haut, on peut compresser un symbole sous forme d'une séquence binaire. En parcourant l'arbre de haut en bas, on peut décompresser une séquence binaire pour restituer un symbole (représenté par le premier nœud feuille trouvé).



Pour la décompression, il faut stocker à la fois la séquence binaire compressée et l'arbre de Huffman qui a rendu la compression possible. Quand le fichier à traiter est trop petit, l'arbre de Huffman peut avoir besoin de davantage d'espace que celui qu'occupent les données compressées, si bien que la compression est inefficace. Le code de Huffman est plus adapté aux fichiers de données de grande dimension.

Garder la mémoire des séquences avec LZW

Le codage de Huffman tire parti des caractères, des chiffres ou des symboles les plus fréquents et réduit leur représentation codée. L'algorithme LZW effectue une tâche similaire mais étend le processus de codage aux séquences de caractères les plus fréquentes. L'algorithme LZW date de 1984. Il a été développé par Abraham Lempel, Jacob Ziv et Terry Welch d'après un algorithme plus ancien, LZ78 (développé en 1978 par Lempel et Ziv seuls). La compression Unix et le format d'image GIF utilisent cet algorithme. LZW tire parti des répétitions, il est donc également idéal pour la compression des documents et des textes, sachant que beaucoup de mots se répètent. Par ailleurs, LZW peut

traiter des données en continu, contrairement à l'algorithme de Huffman : celui-ci a besoin de l'ensemble des données à traiter pour pouvoir constituer son tableau de correspondance.

À mesure que l'algorithme traite le flux de données, il assimile les séquences de caractères et attribue à chaque séquence un code court. Ainsi, quand il retrouve par la suite la même série de caractères, LZW peut les compresser à l'aide d'un codage plus simple. L'aspect intéressant de cet algorithme est qu'il prend pour point de départ une table de symboles constituée de caractères (généralement la table ASCII) et agrandit ce tableau à l'aide des séquences de caractères qu'il apprend des données qu'il compresse.

Par ailleurs, LZW n'a pas besoin de stocker les séquences apprises dans un tableau pour la décompression : il peut les restituer facilement en lisant les données compressées. LZW est capable d'inverser les étapes du processus de compression des données et de codage des séquences, mais cette capacité a son revers : LZW n'est pas toujours très efficace. Il l'est davantage avec de gros paquets de données ou de texte (une caractéristique qu'il partage avec d'autres algorithmes de compression).

LZW n'est pas un algorithme complexe, mais il faut étudier plusieurs exemples pour pouvoir bien le comprendre. Vous trouverez de bons tutoriaux à l'adresse <http://marknelson.us/2011/11/08/lzw-revisited/> et à l'adresse http://www.matthewflickinger.com/lab/whatsinagif/lzw_image_da Le deuxième tutoriel explique comment utiliser LZW pour compresser des images. L'exemple suivant est une application avec Python (vous retrouverez le code complet de cet exemple dans la section LZW du fichier téléchargeable A4D ; 14 ; Compression.ipynb : pour plus de détails, consultez l'Introduction).

```
def lzw_compress(text):
    dictionary = {chr(k): k for k in range(256)}
    encoded = list()
    s = text[0]
    for c in text[1:]:
```

```

        if s+c in dictionary:
            s = s+c
        else:
            print ('> %s' %s)
            encoded.append(dictionary[s])
            print ('trouvé : %s compressé sous la forme
%s' %
                (s,dictionary[s]))
            dictionary[s+c] = max(dictionary.values()) +
1
            print ('Nouvelle séquence %s indexée sous la
forme %s' %
                (s+c, dictionary[s+c]))
            s = c
            encoded.append(dictionary[s])
            print ('trouvé : %s compressé sous la forme %s'
                %(s,dictionary[s]))
            return encoded

```

Dans cet exemple, l'algorithme parcourt le texte en examinant les caractères un à un. Il code d'abord les caractères en utilisant la table de symboles initiale, qui est ici la table ASCII. Le meilleur moyen de comprendre comment fonctionne ce codage consiste à observer une série de messages en output puis d'analyser ce qui s'est passé :

```

text = "ABABCABCABC"
compressed = lzw_compress(text)
print('\nCompressé : %s \n' % compressed)

```

```

> A
trouvé : A compressé sous la forme 65
Nouvelle séquence AB indexée sous la forme 256
> B
trouvé : B compressé sous la forme 66
Nouvelle séquence BA indexée sous la forme 257
> AB
trouvé : AB compressé sous la forme 256
Nouvelle séquence ABC indexée sous la forme 258
> C
trouvé : C compressé sous la forme 67
Nouvelle séquence CA indexée sous la forme 259
> ABC
trouvé : ABC compressé sous la forme 258
Nouvelle séquence ABCA indexée sous la forme 260
trouvé : ABC compressé sous la forme 258

```

Voici un bref résumé de ce que signifient ces messages de sortie :

1. **La première lettre, A, apparaît dans la table de symboles initiale, et l'algorithme lui attribue le code 65.**
2. **La deuxième lettre, B, est différente de A mais apparaît aussi dans la table des symboles initiale, et l'algorithme lui attribue le code 66.**
3. **La troisième lettre est encore un A, par conséquent l'algorithme lit la lettre suivante, qui est un B, et attribue à la combinaison de ces deux lettres AB le code 256.**
4. **La quatrième lettre, C, est différente des deux lettres précédentes et apparaît également dans la table des symboles initiale, l'algorithme lui attribue donc le code 67.**
5. **La lettre suivante est déjà apparue précédemment : c'est un A. Elle est suivie d'un B, si bien que nous avons la combinaison AB, qui apparaît dans la table des symboles. Cependant, la lettre qui suit est un C, ce qui donne une nouvelle séquence à laquelle l'algorithme attribue maintenant le code 258.**
6. **Les trois dernières lettres forment une autre série ABC, par conséquent le programme leur attribue à nouveau le code 258. L'output codé pour la séquence ABABCABCABC est donc**

Compressé : [65, 66, 256, 67, 258, 258]

Toutes ces opérations d'apprentissage et de codage aboutissent à un jeu de données compressées constitué de seulement six codes numériques (qui occupent chacun 8 bits), pour un jeu de données initial de 11 lettres. Le codage donne un bon taux de compression, d'environ la moitié du volume initial des données : $6 / 11 = 0,55$.

Restituer le texte original à partir des données compressées nécessite une procédure inverse, qui tienne compte de la seule

situation dans laquelle le décodage LZW peut échouer à reconstituer la table des symboles lorsqu'une séquence commence et se termine par le même caractère. Ce cas particulier est pris en compte par Python grâce à un bloc d'instructions de type *if-then-else*, de telle sorte que l'algorithme permette de tout coder et décoder en toute sécurité :

```
def lzw_decompress(encoded):
    reverse_dictionary = {k:chr(k) for k in range(256)}
    current = encoded[0]
    output = reverse_dictionary[current]
    print ('Décompression de %s ' % output)
    print ('>%s' % output)
    for element in encoded[1:]:
        previous = current
        current = element
        if current in reverse_dictionary:
            s = reverse_dictionary[current]
            print (' Décompression de %s ' % s)
            output += s
            print ('>%s' % output)
            new_index = max(reverse_dictionary.keys()) +
1
            reverse_dictionary[new_index
            ] = reverse_dictionary[previous] + s[0]
            print ('Nouvelle entrée de dictionnaire %s à
1'index %s' %
                (reverse_dictionary[previous] + s[0],
                new_index))
        else:
            print ('Pas trouvé :',current,'Output :',
                reverse_dictionary[previous
                ] + reverse_dictionary[previous][0])
            s = reverse_dictionary[previous
                ] + reverse_dictionary[previous][0]
            print ('Nouvelle entrée de dictionnaire %s à
1'index %s' %
                (s, max(reverse_dictionary.keys()+1))
            reverse_dictionary[
                max(reverse_dictionary.keys()+1)] = s
            print (' Décompression de %s' % s)
            output += s
            print ('>%s' % output)
    return output
```

L'exécution de la fonction sur la séquence compressée restitue l'information initiale grâce à un parcours de la table de symboles :

```
print ('\nChaîne décompressée : %s' %  
      lzw_decompress(compressed))  
print ('La chaîne initiale était : %s' % text)
```

Décompression de A

> A

Décompression de B

> AB

Nouvelle entrée de dictionnaire AB à l'index 256

Décompression de AB

> ABAB

Nouvelle entrée de dictionnaire BA à l'index 257

Décompression de C

> ABABC

Nouvelle entrée de dictionnaire ABC à l'index 258

Décompression de ABC

> ABABCABC

Nouvelle entrée de dictionnaire CA à l'index 259

Décompression de ABC

> ABABCABCABC

Nouvelle entrée de dictionnaire ABCA à l'index 260

Chaîne décompressée : ABABCABCABC

La chaîne initiale était : ABABCABCABC

PARTIE 5

Traiter des problèmes difficiles

DANS CETTE PARTIE...

Utiliser des techniques de programmation gloutonne pour obtenir des résultats plus vite

Effectuer une programmation dynamique pour exécuter des tâches en utilisant une approche intelligente

Randomiser les résultats pour résoudre des problèmes pour lesquels une approche directe ne serait pas adaptée

Rechercher localement des solutions acceptables pouvant être obtenues rapidement

Utiliser les techniques de programmation linéaire pour exécuter des tâches d'ordonnancement et de planification

Recourir à l'heuristique et interagir avec des robots

Chapitre 15

Travailler avec des algorithmes gloutons

DANS CE CHAPITRE

- » Apprendre à concevoir de nouveaux algorithmes et utiliser des paradigmes de résolution
 - » Expliquer comment un algorithme peut se montrer glouton et donner d'excellents résultats
 - » Construire soi-même un algorithme glouton
 - » Réviser le codage de Huffman et l'illustrer par d'autres exemples classiques
-

Maintenant que vous avez effectué vos premiers pas dans le monde des algorithmes, compris en quoi ils consistent et étudié le tri, la recherche, les graphes et les grandes données, le moment est venu pour vous d'aborder une partie plus générale du livre. Dans cette dernière partie, vous allez vous colleter à des exemples difficiles et découvrir des méthodes algorithmiques générales que vous pourrez utiliser dans différentes circonstances pour résoudre des problèmes dans le monde réel.

Ce chapitre privilégie de nouvelles méthodes pour aller bien au-delà de l'approche « diviser pour régner » qui est la plus couramment utilisée dans les problèmes de tri. Certaines solutions étudiées ici ne sont pas entièrement nouvelles : elles ont déjà été abordées dans les chapitres précédents. Cependant, ce chapitre les

étudie plus en profondeur, dans le cadre des nouveaux *paradigmes* qui y sont illustrés (règles et conditions d'application, méthode générale et étapes vers la solution du problème, analyse de la complexité du problème, limites et restrictions).

Généraliser des solutions et les présenter comme des paradigmes largement applicables ouvre la voie à la résolution des nouveaux problèmes pratiques et entre dans le cadre de l'analyse et de la conception des algorithmes. Le reste de ce livre est consacré aux méthodes générales suivantes :

- » les algorithmes gloutons (expliqués dans ce chapitre) ;
- » la programmation dynamique ;
- » la randomisation, la recherche locale et l'heuristique tournée vers l'avenir ;
- » la programmation linéaire et les problèmes d'optimisation.

Quand faut-il être glouton ?

Confronté à des problèmes difficiles, on ne tarde pas à se rendre compte qu'il n'existe aucune potion magique qui permettrait d'exaucer les vœux ni de solutions miracles pour éloigner le mal. De même, aucune technique algorithmique n'est idéale dans toutes les situations. Il y a toujours un compromis à trouver, et c'est là un principe qui revient souvent dans ce livre. Heureusement, vous avez la possibilité de maîtriser différentes techniques et de les essayer toutes, avec de bonnes chances que l'une d'entre elles donne de bons résultats.

Les algorithmes gloutons sont pratiques pour résoudre un vaste ensemble de problèmes, surtout lorsqu'il est difficile d'élaborer une solution globale. Il est parfois préférable de renoncer à des projets compliqués et d'opter simplement pour la solution la plus abordable, dans la mesure où elle ressemble suffisamment à celle qu'il vous faut. Ce type d'approche à courte vue, consistant à rechercher des solutions faciles, est la caractéristique essentielle

des algorithmes *gloutons*. L'algorithme glouton consiste à aboutir à une solution par étapes séquentielles en prenant à chaque étape une décision basée sur la solution qui est la meilleure sur le moment, sans considération des conséquences ni des implications futures.



Deux éléments sont essentiels pour distinguer un algorithme glouton :

- » À chaque étape, on prend toujours la meilleure décision possible sur la base des conditions du moment.
- » En prenant une série de décisions consécutives selon cette règle, on espère aboutir finalement à la meilleure solution.

Les algorithmes gloutons sont simples, intuitifs, courts et rapides, car ils fonctionnent généralement en *temps linéaire* (leur temps d'exécution est proportionnel au nombre de données fournies en entrée). Ils n'offrent malheureusement pas la meilleure solution à tous les problèmes, mais lorsqu'ils sont appropriés à la situation, ils donnent rapidement les meilleurs résultats. Même lorsqu'ils ne fournissent pas les meilleures réponses, ils peuvent apporter une solution qui, sans être optimale, peut être acceptable ou servir de point de départ à une amélioration par une autre méthode algorithmique.

Il est intéressant de remarquer que le fonctionnement des algorithmes gloutons ressemble à la façon dont l'être humain résout des problèmes simples sans avoir besoin de beaucoup solliciter la matière grise, ou avec des informations incomplètes. Quand on doit rendre la monnaie, par exemple, on utilise naturellement une méthode gloutonne. Le problème du *rendu de monnaie* consiste à délivrer une somme donnée (la monnaie à rendre) en utilisant le plus petit nombre de billets et de pièces, compte tenu de plusieurs combinaisons possibles. L'exemple suivant, avec Python, montre comment le problème du rendu de monnaie peut être résolu par un algorithme glouton. Dans cet

exemple, on utilise des billets de 5, 10, 20, 50 et 100 euros, mais aucune pièce.

```
def change(to_be_changed, denomination):
    resulting_change = list()
    for bill in denomination:
        while to_be_changed >= bill:
            resulting_change.append(bill)
            to_be_changed = to_be_changed - bill
    return resulting_change, len(resulting_change)
```

```
currency = [100, 50, 20, 10, 5]
amount = 365
print ('Rendu : %s (avec %i billets)'
      % (change(amount, currency)))
```

Rendu : [100, 100, 100, 50, 10, 5] (avec 6 billets)

L'algorithme, intégré à la fonction `change()`, passe en revue les coupures disponibles, de la plus grande à la plus petite. Il utilise les plus gros billets disponibles pour rendre la monnaie jusqu'à ce que la somme due soit inférieure à la coupure courante. Il passe alors à la coupure suivante et exécute la même tâche, et ainsi de suite jusqu'à ce qu'il atteigne la plus petite coupure. Ainsi, `change()` délivre toujours le plus gros billet possible compte tenu de la somme à rendre (c'est le principe même de l'algorithme glouton).

Les algorithmes gloutons sont particulièrement prisés pour résoudre les problèmes de programmation dans le temps, de cache optimale, et de compression à l'aide du codage de Huffman. Ils conviennent également pour les problèmes de graphe. Ainsi, par exemple, les algorithmes de Kruskal et de Prim pour trouver un arbre couvrant de coût minimum et l'algorithme de Dijkstra pour trouver le plus court chemin sont des algorithmes gloutons (pour plus de détails, voir [Chapitre 9](#)). L'approche gloutonne peut aussi permettre d'obtenir une solution non optimale, mais constituant une première approximation acceptable, au problème du voyageur de commerce, et elle permet de résoudre le problème du sac à dos quand les quantités ne sont pas discrètes (ces deux problèmes sont abordés au [Chapitre 16](#)).

L'avidité, c'est bien

Il n'est pas surprenant qu'une approche gloutonne soit aussi efficace pour résoudre le problème du rendu de monnaie. En effet, pour certains problèmes, une approche prévoyante n'est pas utile : la solution est l'aboutissement d'une série d'étapes (des décisions successives), et à chaque étape la bonne décision est toujours celle qui est la meilleure selon un critère initialement choisi.

La gloutonnerie, ou si l'on préfère, l'avidité, est aussi une approche très humaine (et efficace) pour la résolution des problèmes économiques. Dans le film *Wall Street* (1987), Gordon Gecko, le personnage principal, déclare que « l'avidité, c'est bien ». L'avidité (non pas au sens moral du terme, mais au sens d'agir en vue de maximiser des objectifs particuliers, comme le fait un algorithme glouton) est au cœur même de l'économie néoclassique. Des économistes comme Adam Smith, au XVIII^e siècle, ont considéré que la poursuite des intérêts personnels (hors de toute vision globale) était grandement avantageuse pour la société dans sa globalité et que c'était là la clé de la prospérité (la théorie de la « main invisible » : <https://plus.maths.org/content/adam-smith-and-invisible-hand>).

Le principe de fonctionnement de l'algorithme glouton (et les conditions dans lesquelles il donne satisfaction) ne présente aucune difficulté, et se résume aux quatre étapes suivantes :

- 1. On divise le problème en plusieurs problèmes partiels. La somme (ou autre combinaison) de ces problèmes partiels donne la bonne solution. De ce point de vue, un algorithme glouton n'est pas très différent d'un algorithme conçu selon le principe « diviser pour régner » (comme Quicksort ou Mergesort, ces deux exemples étant abordés au [Chapitre 7](#)).**
- 2. L'exécution réussie de l'algorithme dépend de l'exécution réussie de chaque étape. On parle de propriété de sous-structure optimale, sachant qu'une**

solution optimale n'est constituée que de solutions partielles optimales.

- 3. Afin de réussir chaque étape, l'algorithme ne prend en compte les données d'entrée qu'à l'étape concernée. En d'autres termes, c'est la situation immédiate (résultat des décisions précédentes) qui détermine la décision prise par l'algorithme, sans prise en compte des conséquences. Cette absence totale de stratégie globale est la propriété du choix glouton, l'avidité à chaque phase étant la condition suffisante pour la réussite finale. À titre d'analogie, tout se passe comme si l'on était sûr de gagner aux échecs en ne voyant jamais plus loin que le coup joué.**
- 4. La propriété du choix glouton donnant un bon espoir de succès, l'algorithme glouton n'obéit pas à une règle complexe de décision. En effet, il ne tient compte que des éléments d'input disponibles à chaque phase. Il n'est pas nécessaire de calculer les implications possibles des décisions : par conséquent, la complexité des calculs est linéaire $O(n)$, dans le pire des cas. Les algorithmes gloutons sont prisés parce qu'ils constituent un moyen simple de résoudre des problèmes complexes, alors que les autres types d'algorithmes analysent le problème trop en profondeur et présentent un temps d'exécution trop long.**

Garder la maîtrise des algorithmes gloutons

Face à un problème difficile, on trouve aisément une solution gloutonne en passant par les quatre étapes décrites dans la section précédente. Il suffit de diviser le problème en phases et de déterminer la règle gloutonne à appliquer à chaque étape :

- » Choisissez votre règle de décision (déterminez la méthode la plus simple et la plus rapide).
- » Lancez-vous dans la résolution du problème en appliquant votre règle de décision.
- » Notez (si nécessaire) le résultat de votre décision et déterminez l'état de la résolution du problème.
- » Appliquez la même méthode de façon répétitive à chaque étape, jusqu'à la conclusion.

Quelle que soit la manière dont vous appliquez ces étapes, vous devez déterminer si vous atteignez votre objectif en vous en tenant à une série de décisions à courte vue. Cette approche est efficace pour résoudre certains problèmes, et parfois pour certains cas particuliers, mais elle n'est pas adaptée à la résolution de tous les problèmes. Le problème de rendu de monnaie, par exemple, se résout très bien ainsi lorsqu'il s'agit de monnaie américaine, mais les résultats sont peu satisfaisants avec d'autres monnaies. Prenons une monnaie fictive dont l'unité sera appelée le crédit (un terme souvent employé dans les jeux et les fictions) avec des coupures de 1, 15 et 25 crédits. L'algorithme précédent ne donnera pas la solution optimale pour un rendu de monnaie de 30 crédits :

```
print ('Rendu : %s (avec %i billets)'
      % (change(30, [25, 15, 1])))
```

Rendu : [25, 1, 1, 1, 1, 1] (avec 6 billets)

À l'évidence, la solution optimale était de rendre deux billets de 15 crédits, mais l'algorithme, qui ne voit pas assez loin, commence par la plus grande coupure disponible (25 crédits) puis utilise cinq billets de 1 crédit pour les 5 crédits qu'il reste à rendre.



Certains cadres mathématiques complexes, les *matroïdes* (pour plus de détails, lire l'article sur la page <https://jeremykun.com/2014/08/26/when-greedy-algorithms-are->

[perfectthe-matroid/](#)), permettent de vérifier s'il est possible d'utiliser un algorithme glouton pour la résolution optimale d'un problème particulier. S'il est possible de formaliser un problème en utilisant un matroïde, un algorithme glouton donnera un résultat optimal. Pour certains problèmes, cependant, il existe une solution gloutonne optimale mais qui n'est pas compatible avec le cadre du matroïde (ces structures peuvent être suffisantes sans être nécessaires pour une solution gloutonne optimale, comme l'explique l'article de la page <http://csttheory.stackexchange.com/questions/21367/does-every-greedy-algorithm-have-a-matroid-structure>).

L'utilisateur d'un algorithme glouton doit savoir que les algorithmes gloutons sont efficaces mais ne donnent pas toujours les meilleurs résultats possibles. Quand c'est le cas, c'est parce que le problème se limite à des exemples connus ou parce que le problème est compatible avec le cadre mathématique du matroïde. Même lorsque l'algorithme glouton est ce qui convient le mieux dans un certain contexte, une situation différente peut changer la donne, si bien que les solutions seront seulement bonnes ou acceptables. Dans de nombreux cas, les résultats seront seulement bons ou acceptables, car souvent les algorithmes gloutons ne font pas mieux que les autres solutions :

- » La résolution du problème de rendu de monnaie, précédemment dans ce chapitre, montre qu'un changement peut entraîner le non-fonctionnement de l'algorithme glouton.
- » Le problème de programmation dans le temps (présenté dans la section « Trouver l'utilité qu'il peut y avoir à être glouton », plus loin dans ce chapitre) illustre la parfaite efficacité d'une solution avec un seul ouvrier, mais il ne faut pas s'attendre à ce qu'elle soit viable avec deux ou plusieurs ouvriers.
- » L'algorithme du plus court chemin de Dijkstra ne fonctionne que si les sommets sont affectés d'un poids

positif (les poids négatifs feront boucler indéfiniment l'algorithme autour de certains sommets).

Montrer qu'un algorithme glouton est la meilleure solution est une tâche difficile, qui exige des connaissances solides en mathématiques. Autrement, vous pouvez élaborer une démonstration de façon plus empirique en testant l'algorithme glouton comparativement à l'une des solutions suivantes :

- » Comparativement à une solution optimale connue, lorsque l'algorithme glouton produit la solution optimale ou lorsque vous pouvez changer cette solution en échangeant ses éléments contre une solution optimale équivalente (sans aucune perte de performance et sans aucun échec). Quand le résultat d'un algorithme glouton est équivalent au résultat d'une solution optimale, on sait que la solution gloutonne est tout aussi valable et convient parfaitement (c'est la *preuve par l'échange*).
- » Comparativement à un autre algorithme quand on remarque que l'algorithme glouton conserve une *longueur d'avance*, c'est-à-dire quand à chaque étape, il donne toujours une meilleure solution que l'autre algorithme.



Même en considérant que la détermination de la solution optimale par un algorithme glouton est plus l'exception que la règle, les solutions gloutonnes sont souvent meilleures que les autres. On n'obtient pas toujours la solution optimale, mais on obtient des résultats assez acceptables pour constituer (au minimum) un bon point de départ, c'est pourquoi il convient, face à un nouveau problème, de commencer par essayer une solution de ce type.

Étudier des problèmes NP-complets

Généralement, on envisage un algorithme glouton parce que les autres méthodes ne délivrent pas la solution voulue dans un délai acceptable. L'algorithme glouton est une approche qui convient lorsqu'il y a un certain nombre de choix à faire, et lorsqu'il faut

les combiner. Quand le nombre de combinaisons possibles augmente, tout devient très complexe et même l'ordinateur le plus puissant ne peut plus fournir une réponse dans un délai raisonnable. Pour reconstituer un puzzle, par exemple, on pourrait, bien sûr, chercher toutes les façons d'assembler les pièces, mais il est plus raisonnable de commencer par choisir un endroit précis et chercher la pièce qui s'y adapte le mieux. Ainsi, on consacre son temps à chercher la pièce la plus adaptée, mais ensuite on n'a plus besoin de s'occuper de cette même zone, si bien que le nombre total de pièces à assembler se réduit à chaque itération.

Les problèmes de puzzle, dans lesquels le nombre de décisions possibles peut devenir considérable, sont plus fréquents qu'on pourrait le penser. Certains problèmes de ce type ont déjà été résolus, mais bien d'autres sont toujours en attente de l'être, et il n'est même pas (encore) possible de les mettre sous une forme telle que l'on sache les résoudre. En attendant qu'un chercheur ait assez de génie pour trouver une solution générique, l'approche gloutonne reste le moyen le plus facile d'envisager ces problèmes, à condition d'accepter l'idée de ne pas toujours obtenir la meilleure solution et de se contenter d'une solution à peu près acceptable (dans de nombreux cas).

Ces problèmes difficiles peuvent avoir des caractéristiques variées et relever de domaines différents. Parmi les exemples de problèmes difficiles, on peut citer le dépliement des protéines (pouvant permettre de guérir un cancer) et le décodage de mots de passe sécurisés, par exemple avec le système de cryptage RSA (<http://blogs.ams.org/mathgradblog/2014/03/30/rsa/>). Dans les années soixante, des chercheurs avaient trouvé une propriété commune à tous ces problèmes, le fait qu'ils présentent la même difficulté de résolution. C'est ce que l'on a appelé, dans la théorie de la complexité, le *problème NP-complet* (NP signifiant non déterministe polynomial). De ce point de vue, ces problèmes se distinguent des autres en ce qu'il n'est pas encore possible de leur trouver une solution dans un délai raisonnable, c'est-à-dire en temps polynomial.

Le *temps polynomial* signifie que le temps utilisé par l'algorithme

se calcule en puissances du nombre d'entrées (classe de problèmes P). Le temps linéaire est un temps polynomial, puisque sa formule est $O(n^1)$. La complexité quadratique $O(n^2)$ et la complexité cubique $O(n^3)$ entrent aussi dans la catégorie du temps polynomial, et bien que leur croissance soit très rapide, elles ne se comparent pas à la complexité NP-complet, qui est généralement un *temps exponentiel*, c'est-à-dire $O(c^n)$. Dans le cas de la complexité exponentielle, il est impossible de trouver une solution raisonnable par la force brute, quel que soit le problème. En effet, si n est assez grand, le nombre de solutions à essayer peut vite dépasser le nombre d'atomes présents dans l'univers connu. L'espoir des spécialistes de l'algorithmique est que quelqu'un trouve un jour un moyen de résoudre un de ces problèmes, ce qui ouvrirait la porte à la résolution de tous les problèmes NP-complets à la fois. Résoudre les problèmes NP-complets est un des « Problèmes du prix du millénaire » proposés par l'Institut de mathématiques Clay, qui offre une récompense d'un million de dollars à quiconque saura élaborer une solution (<http://www.claymath.org/millenniumproblems/p-vs-np-problem>).



NP est une catégorie générale de problèmes d'algorithmique incluant aussi bien les problèmes de type P que NP-complet. De façon générale, les problèmes NP sont difficiles (ils nécessitent la mise au point d'un algorithme intelligent). Les problèmes de type P peuvent être résolus en temps polynomial ; les problèmes de type NP-complet sont si difficiles à résoudre que les algorithmes nécessaires fonctionnent en temps exponentiel. Heureusement, si vous pensez avoir une solution à un problème NP-complet, vous pouvez facilement en vérifier la validité.



Peut-être ne résoudra-t-on aucun problème NP-complet en utilisant un algorithme spécifiquement conçu pour trouver une solution optimale. Néanmoins, il reste possible de trouver une solution raisonnable à l'aide d'algorithmes gloutons.

Trouver l'utilité qu'il peut y avoir à

être glouton

Après avoir abordé les algorithmes gloutons de façon générale, il est instructif d'en étudier certains dans le détail, d'en comprendre le fonctionnement et de déterminer comment réutiliser leurs méthodes pour résoudre d'autres problèmes. Les sections qui suivent étudient l'algorithme de codage de Huffman en vue de mieux comprendre comment il fonctionne et de créer de nouveaux systèmes de codage efficaces. Ces sections décrivent aussi le fonctionnement du cache d'un ordinateur (le cache est un algorithme que tous les ordinateurs utilisent). Par ailleurs, vous allez découvrir une méthode pour ordonner correctement les tâches dans le temps afin de respecter les échéances et les priorités. La production de biens matériels est très dépendante des algorithmes gloutons qui servent à programmer les ressources et les activités. Ces algorithmes constituent généralement le cœur des logiciels de planification des ressources de production et ils sont indispensables au bon fonctionnement des usines (<http://searchmanufacturingerp.techtarget.com/definition/Material-requirements-planning-MRP>).

Organiser des données en cache

Souvent, un ordinateur doit traiter un certain nombre de fois les mêmes données. Or, aller chercher ces données sur un disque ou sur Internet prend du temps et représente un coût en termes de temps machine. C'est pourquoi il est utile de stocker localement les données qui sont souvent utilisées, dans un espace plus facilement accessible (et si elles sont déjà prétraitées, c'est encore mieux). C'est à cela que sert le *cache*, constitué généralement d'une série d'emplacements de mémoire ou d'un espace réservé sur le disque.

Quand vous parcourez l'historique de votre navigation sur l'Internet, par exemple, vous pouvez remarquer qu'une partie seulement du trafic est constituée de nouveaux sites Web, tandis

que vous avez passé bien plus de temps et cherché bien plus de pages sur des sites que vous avez l'habitude de visiter. Le stockage en cache de certaines parties de ces sites (comme l'en-tête, le fond, certaines images, et des pages dont le contenu change peu souvent) peut véritablement améliorer votre expérience Web en réduisant le volume de données à télécharger à nouveau. Tout ce dont vous avez besoin, ce sont les nouvelles données de l'Internet, dans la mesure où ce que vous voulez voir se trouve déjà en grande partie quelque part dans votre ordinateur (le cache d'un navigateur est un répertoire de disque.)

Le problème n'est pas nouveau. Dans les années soixante, László Bélády, un informaticien hongrois qui travaillait chez IBM Research, avait émis l'hypothèse que le meilleur moyen de stocker l'information dans un ordinateur pour pouvoir la réutiliser rapidement était de savoir quelles données seraient nécessaires dans le futur et pendant combien de temps. Or, il n'est pas possible de mettre en application de telles prévisions, l'utilisation de l'ordinateur étant souvent imprédictible et non pas prédéterminée.

Cependant, sur le plan du principe, cette idée d'anticiper l'avenir peut inspirer une stratégie optimale de remplacement, un choix glouton fondé sur l'idée de conserver les pages que l'on pense utiliser bientôt, en se fondant sur les requêtes précédentes adressées au cache. L'*algorithme optimal de remplacement des lignes de cache* de Bélády fonctionne selon le principe de l'algorithme glouton : supprimer du cache les données dont la prochaine utilisation devrait être plus éloignée dans le futur, afin de minimiser le risque de supprimer des données dont l'utilisateur ne va pas tarder à avoir besoin. L'algorithme procède comme suit :

- 1. Remplir le cache de l'ordinateur en enregistrant les données de chaque requête lancée. Quand le cache est saturé, et pas avant, on commence à supprimer des données afin de faire de la place pour de nouvelles données.**

2. **Définir une méthode pour déterminer l'utilisation récente. À cet effet, l'algorithme peut utiliser les dates ou un système d'indicateurs de mémoire (détermination des indicateurs récemment utilisés et suppression des indicateurs au bout d'un certain temps).**
3. **Chaque fois que de nouvelles données doivent y être stockées, on supprime du cache les données qui n'ont pas été utilisées récemment. L'algorithme les sélectionne au hasard parmi les données non utilisées.**

Ainsi, par exemple, supposons que votre cache ne soit constitué que de quatre emplacements de mémoire et qu'il soit rempli par quatre lettres de l'alphabet arrivant dans l'ordre suivant :

ABCD

Quand une nouvelle lettre doit être traitée, par exemple la lettre *E*, l'ordinateur fait de la place en supprimant une des lettres les moins susceptibles de faire l'objet d'une requête. Dans cet exemple, les bons candidats sont *A*, *B* et *C* (*D* étant l'ajout le plus récent). L'algorithme choisira un emplacement au hasard et supprimera du cache la donnée correspondante afin que le *E* puisse la remplacer.

En concurrence pour les ressources

Quand il s'agit d'atteindre un objectif, comme créer un service ou produire un objet matériel, un problème courant est la programmation dans le temps de plusieurs activités qui exigent chacune un accès exclusif aux ressources. Ces ressources peuvent être le temps ou une machine. De telles situations abondent dans le monde réel, depuis le calendrier des cours à l'université jusqu'à la gestion des approvisionnements d'une armée, en passant par l'assemblage d'un produit complexe comme une automobile ou l'organisation de sessions de travail informatique dans un centre de données. Dans ces situations, les objectifs sont toujours les mêmes :

- » Obtenir que la plus grande quantité de travail possible soit accomplie dans un certain délai.
- » Gérer les tâches aussi rapidement que possible, en moyenne.
- » Respecter certaines priorités strictes (délais fermes).
- » Respecter certaines indications de priorités (délais souples).

La programmation des tâches comporte deux catégories :

- » Les tâches qu'il est difficile d'accomplir correctement et qui doivent être confiées à des algorithmes élaborés.
- » Les tâches qui sont plus faciles à gérer et qui peuvent être accomplies par de simples algorithmes gloutons.

La plupart des tâches d'organisation peuvent être effectuées par des algorithmes gloutons. Faire exécuter les tâches le plus rapidement possible, par exemple, est une exigence courante dans le domaine de la production industrielle et dans le secteur des services, lorsque chaque tâche répond aux besoins d'un client et lorsque l'on cherche à satisfaire au mieux tous les clients. Voici comment vous pouvez déterminer un contexte pour ce type d'algorithme :

- » Vous disposez d'une seule machine (ou d'un seul ouvrier) pour traiter les commandes.
- » Les commandes arrivent par paquets, si bien que vous êtes confronté à un choix étendu.
- » Certaines commandes sont plus longues que d'autres, et le temps d'exécution sera différent pour chaque commande.

Supposons que vous receviez de quatre clients différents quatre tâches à exécuter, et que l'exécution de ces tâches nécessite respectivement huit heures, quatre heures, douze heures et trois heures. Bien que le temps total d'exécution ne change pas, l'ordre dans lequel les tâches seront exécutées aura un impact sur le délai

pour terminer le travail et sur le temps que chaque client devra attendre avant que sa commande soit exécutée. Les sections qui suivent étudient différentes méthodes pour répondre aux besoins des clients compte tenu d'objectifs spécifiques.

Assurer la satisfaction des clients

Dans toute activité commerciale, il importe avant tout d'assurer la satisfaction des clients. Si l'on exécute les tâches dans l'ordre dans lequel elles sont présentées, il faudra $8 + 4 + 12 + 3 = 27$ heures pour que tout soit terminé. Cependant, le premier client devra attendre 8 heures, et le dernier devra attendre 27 heures. En l'occurrence, la première tâche sera terminée au bout de 8 heures, la deuxième au bout de $8 + 4 = 12$ heures, la troisième au bout de $8 + 4 + 12 = 24$ heures, et la dernière au bout de $8 + 4 + 12 + 3 = 27$ heures.

Si votre objectif est que tous vos clients soient contents et satisfaits, mieux vaudrait minimiser le temps d'attente moyen. En l'occurrence, ce temps d'attente moyen est $(8 + 12 + 24 + 27) / 4 = 17,75$ heures. Afin de réduire ce temps d'attente moyen, vous pourriez commencer par simuler toutes les possibilités d'ordre d'exécution et recalculer cette estimation. Cette méthode est envisageable s'il s'agit d'un petit nombre de tâches à exécuter sur une seule machine, mais lorsque vous avez des centaines de tâches à exécuter sur plusieurs machines, cela devient un problème de calcul très compliqué. Un algorithme glouton vous permet de sauver la mise sans nécessiter un gros travail de planification : il vous suffit d'exécuter en priorité la tâche la plus courte. Le temps d'attente moyen ainsi obtenu sera le plus réduit possible : $(3 + (3 + 4) + (3 + 4 + 8) + (3 + 4 + 8 + 12)) / 4 = 13$ heures.



Pour obtenir le temps d'attente moyen, on calcule la moyenne des temps d'exécution cumulés. Si l'on calculait la moyenne des temps d'exécution, on obtiendrait la durée moyenne d'une tâche, mais ce chiffre ne serait pas représentatif du temps d'attente du client.

Le principe de l'algorithme glouton est simple : sachant qu'on additionne les temps d'exécution cumulés, en exécutant d'abord les tâches les plus longues, on obtiendrait la somme des cumuls les plus élevés. Au contraire, en commençant par les tâches les plus courtes, les premiers termes de l'addition sont constitués des temps d'exécution les plus courts, ce qui influence favorablement le temps d'attente moyen (et donc le niveau de satisfaction des clients).

Respecter les délais

Parfois, il ne s'agit pas seulement de limiter le temps d'attente des clients, mais aussi de respecter leurs contraintes de temps : vous avez alors des échéances dont vous devez tenir compte. Dans ce cas, l'algorithme glouton n'est plus le même. Il ne s'agit plus de commencer par la tâche la moins longue, mais par la tâche pour laquelle la date butoir est la plus proche, sachant que le plus tôt est le mieux. Ce problème est celui des délais impératifs, et il n'est pas toujours possible de le résoudre (il arrive que des échéances ne puissent tout simplement pas être respectées).



Si vous ne pouvez pas résoudre le problème avec un algorithme glouton, il vous faut admettre qu'aucune solution n'existe. Quand une échéance stricte ne peut pas être respectée, vous pouvez essayer de résoudre le problème avec des échéances souples, c'est-à-dire en respectant plutôt les priorités (en exécutant d'abord certaines tâches, en fonction de leur degré de priorité).

Dans cet exemple, il y a à la fois la durée de chaque tâche, comme dans la section précédente, et une valeur (un poids) représentant l'importance de la tâche (un poids plus grand signifie une plus grande priorité). Le problème est le même, sauf que cette fois il s'agit de minimiser le temps d'exécution moyen pondéré. Pour atteindre cet objectif, on attribue aux tâches des rangs de priorité en divisant les durées par les poids, et l'on exécute d'abord les tâches dont le rang est le plus faible. La tâche qui a le rang le moins élevé est soit une tâche très prioritaire, soit une tâche très courte.

Ainsi, pour reprendre l'exemple précédent, nous avons maintenant des tuples constitués de poids et de durées : (40,8), (30,4), (20,12), (10,3), le nombre 40 dans le premier tuple étant un poids et le nombre 8 une durée. En divisant chaque durée par le poids correspondant, on obtient les rangs de priorité suivants : 0,20 ; 0,13 ; 0,60 ; 0,30. On commence par le rang le moins élevé, et en ajoutant à chaque fois le rang de priorité le moins élevé, on obtient l'ordre d'exécution qui minimise les durées et respecte les priorités : (30,4), (40,8), (10,3), (20,12).

Revoir et corriger le codage de Huffman

Comme on l'a vu au chapitre précédent, le codage de Huffman peut représenter le contenu d'une série de données sous une forme plus compacte en tirant parti du fait que certaines données (par exemple certains caractères de l'alphabet) apparaissent plus souvent dans le flux de données. En attribuant aux données des codes de longueur variable (plus courts pour les caractères les plus fréquents, plus longs pour les moins fréquents), on réduit l'espace utilisé pour les stocker. Robert M. Fano (le professeur de Huffman) et Claude Shannon avaient déjà envisagé une telle méthode de compression, mais sans trouver un moyen efficace de déterminer une solution de codage avec laquelle il serait impossible de confondre un caractère avec un autre.



Des codes sans préfixe sont nécessaires pour éviter les erreurs lors du décodage du message. Cela signifie qu'aucun codage de bits déjà utilisé ne doit servir de point de départ pour un autre codage de bits. Huffman avait trouvé une solution simple et pratique pour utiliser des codes sans préfixe à l'aide d'un algorithme glouton. Cette solution consiste à transformer l'arbre équilibré initial (l'arbre équilibré est une structure de données étudiée au [Chapitre 6](#)) correspondant au codage sur une longueur fixe en arbre non équilibré ([Figure 15-1](#)).



L'arbre non équilibré présente une caractéristique particulière, que de chaque nœud ne part qu'une branche aboutissant à d'autres nœuds et branches, tandis que l'autre branche se termine par un caractère codé. Cette caractéristique garantit qu'aucune séquence de codage déjà utilisée ne peut commencer une nouvelle séquence (graphiquement, une branche qui se termine par un caractère codé est un cul-de-sac).

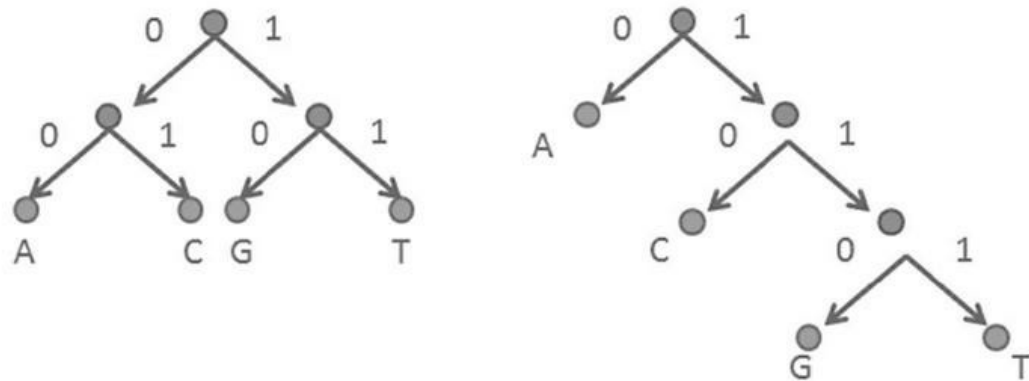


FIGURE 15-1 D'un arbre équilibré (à gauche) à un arbre non équilibré (à droite).

Outre la représentation graphique de la structure non équilibrée, un algorithme glouton peut aussi construire un arbre non équilibré. L'idée est de créer la structure à partir de la racine, en commençant par les caractères les moins fréquemment utilisés. L'algorithme crée les niveaux supérieurs de l'arbre en ajoutant peu à peu les caractères moins fréquents jusqu'à ce qu'il n'y ait plus de caractères et que l'on atteigne le sommet.

À titre de démonstration du principe de gloutonnerie sur lequel repose l'algorithme, cette section présente un exemple de code Python ayant pour thème l'ADN. L'ADN est représenté par une séquence formée des lettres A, C, T et G (les quatre nucléotides qui sont présents chez tous les êtres vivants). Une bonne idée est d'utiliser seulement deux bits pour représenter chacune des quatre lettres, ce qui est déjà une méthode appréciable pour économiser de la mémoire par rapport à l'utilisation d'un codage en ASCII complet (qui se fait sur 7 bits au minimum).

Les nucléotides ne sont pas répartis uniformément. Leur distribution varie selon les gènes étudiés. Le tableau suivant représente un gène avec une distribution inégale, avec une prédominance des nucléotides A et C.

Nucléotides	Pourcentage	Codage sur longueur fixe	Codage de Huffman
A	40,5 %	00	0
C	29,2 %	01	10
G	14,5 %	10	110
T	15,8 %	11	111
Moyenne pondérée du nombre de bits		2,00	1,90

En multipliant le nombre de bits dans les deux codages par leur pourcentage et en additionnant tout, on obtient la moyenne pondérée du nombre de bits utilisés, avec chaque méthode. En l'occurrence, le résultat est 1,9 pour le codage de Huffman contre 2,0 pour le codage sur longueur fixe. Dans cet exemple, on économise donc 5 % du nombre de bits. On pourrait économiser davantage d'espace encore avec des gènes qui présenteraient une distribution encore plus inégale en faveur d'un nucléotide.

L'exemple suivant génère une séquence d'ADN aléatoire et montre comment le code produit systématiquement le codage (si l'on change la valeur de départ, la génération aléatoire de séquences d'ADN peut aboutir à un résultat différent, aussi bien dans la distribution des nucléotides que dans le codage de Huffman).

```
from heapq import heappush, heappop, heapify
from collections import defaultdict, Counter
from random import shuffle, seed
generator = ["A"]*6+["C"]*4+["G"]*2+["T"]*2
text = ""
seed(4)
```

```

for i in range(1000):
    shuffle(generator)
    text += generator[0]

print(text)
frequencies = Counter(list(text))
print(frequencies)

CAACCCCGACACGCCTCCATAGCCACAACAAGCAAAAAAGGC ...
Counter({'A': 405, 'C': 292, 'T': 158, 'G': 145})

```

Après avoir préparé la compression des entrées de données, le programme prépare une structure de tas (pour plus de détails, voir la section « Effectuer des recherches dans un domaine particulier en utilisant un tas binaire » du [Chapitre 7](#)) afin d'organiser efficacement les résultats tout au long des étapes suivies par l'algorithme. Les éléments du tas contiennent la fréquence des nucléotides, les caractères des nucléotides et le codage. Avec une complexité linéarithmique $O(n * \log(n))$, le tas est la structure appropriée pour ordonner les résultats et pour que l'algorithme retire rapidement les deux plus petits éléments.

```

heap = ([[freq, [char, "]] for char, freq in
        frequencies.items()])
heapify(heap)
print(heap)
[[145, ['G', '']], [158, ['T', '']], [405, ['A', '']],
 [292, ['C', '']]]

```

L'algorithme sélectionne dans le tas les nucléotides dont les fréquences sont les moins élevées (le choix glouton). Il regroupe ces nucléotides sous la forme d'un nouvel élément qui va remplacer les deux précédents. Le processus continue jusqu'à ce que la dernière agrégation réduise le nombre d'éléments du tas à un.

```

iteration = 0
while len(heap) > 1:
    iteration += 1
    lo = heappop(heap)
    hi = heappop(heap)
    print ('Étape %i 1er :%s 2d :%s' % (iteration,

```

```

lo,hi))
    for pair in lo[1:]:
        pair[1] = '0' + pair[1]
    for pair in hi[1:]:
        pair[1] = '1' + pair[1]
    heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

```

```

Étape 1 1er :[145, ['G', '']] 2d :[158, ['T', '']]
Étape 2 1er :[292, ['C', '']] 2d :[303, ['G', '0'],
['T', '1']]
Étape 3 1er :[405, ['A', '']] 2d :[595, ['C', '0'],
['G', '10'], ['T', '11']]

```

À mesure que les nucléotides sont regroupés et qu'ainsi se constituent les différents niveaux de l'arbre non équilibré, leur codage de Huffman est modifié de façon systématique : un 0 est ajouté devant le code de l'agrégat le moins fréquent et un 1 est ajouté devant le code du second agrégat le moins fréquent. De cette manière, l'algorithme reproduit bien la structure d'arbre non équilibré illustrée précédemment.

```

tree = sorted(heapop(heap)[1:], key=lambda p:
(len(p[-
    1]), p))
print ("Symbole\tPoids\tCode")
for e in tree:
    print ("%s\t%s\t%s" % (e[0], frequencies[e[0]],
e[1]))

```

Symbole	Poids	Code
A	4050	
C	29210	
G	145110	
T	158111	

L'étape finale est l'affichage du résultat, dont les éléments sont triés selon le codage par bit, avec la table de symboles finale créée.

Chapitre 16

Utiliser la programmation dynamique

DANS CE CHAPITRE

- » Comprendre le sens du mot dynamique lorsqu'il s'agit de programmation
 - » Faire un usage efficace de la mémorisation pour les besoins de la programmation dynamique
 - » Découvrir l'utilité que peut avoir le problème du sac à dos pour l'optimisation
 - » Travailler sur le problème NP-complet du voyageur de commerce
-

Plutôt que de recourir à la force brute, laquelle consiste à essayer toutes les solutions possibles, les algorithmes gloutons fournissent une réponse qui est rapide et souvent satisfaisante. En fait, un algorithme glouton peut parfois résoudre entièrement le problème. Cependant, les algorithmes gloutons sont aussi limités car ils prennent des décisions qui ne tiennent pas compte des conséquences de leurs choix. Le [Chapitre 15](#) explique qu'on ne peut pas toujours résoudre un problème à l'aide d'un algorithme glouton. Un algorithme peut prendre à une certaine étape une décision apparemment optimale, mais qui apparaîtra par la suite limitative et sous-optimale dans le cadre de la recherche de la meilleure solution. Un meilleur algorithme, qui ne sera pas fondé sur l'approche gloutonne, pourra réviser les décisions

passées ou anticiper le fait qu'une décision apparemment bonne ne soit pas aussi prometteuse qu'elle le semblait. Cette approche est celle que suit la programmation dynamique.

La *programmation dynamique* est une méthode algorithmique conçue dans les années cinquante par Richard Ernest Bellman (un mathématicien également connu pour d'autres avancées dans le domaine des mathématiques et des algorithmes, voir https://fr.wikipedia.org/wiki/Richard_Bellman) qui teste davantage de solutions que la méthode gloutonne correspondante. Tester davantage de solutions donne la possibilité de réfléchir aux conséquences des décisions. La programmation dynamique permet d'éviter de devoir effectuer de fastidieux calculs, grâce à un système de cache ingénieux appelé la *mémoïsation*, un terme défini plus loin dans ce chapitre (un *cache* étant un système de stockage de données).

Ce chapitre vous propose davantage qu'une simple définition de la programmation dynamique. Il vous explique aussi pourquoi la programmation dynamique s'appelle ainsi et comment transformer n'importe quel algorithme (et plus particulièrement les algorithmes récurrents) en programmation dynamique à l'aide de Python et de ses décorateurs (des outils puissants qui vous permettent de modifier une fonction sans devoir en réécrire le code). En outre, vous allez découvrir des applications de la programmation dynamique pour optimiser les ressources et les rendements, pour trouver des raccourcis entre deux points et pour comparer les chaînes de caractères de façon approximative. La programmation dynamique fournit une approche naturelle de la gestion des divers problèmes que vous pouvez rencontrer au cours de votre périple dans le monde des algorithmes.

Expliquer la programmation dynamique

La programmation dynamique est aussi efficace qu'un algorithme exhaustif (elle donne donc des solutions correctes), mais elle est souvent aussi efficace qu'une solution approximative (le temps de calcul des algorithmes de programmation dynamique étant souvent polynomial). Son fonctionnement a quelque chose de magique, sachant que souvent, pour la solution dont on a besoin, il faut que l'algorithme exécute les mêmes calculs un certain nombre de fois. En modifiant l'algorithme pour le rendre dynamique, on peut enregistrer les résultats des calculs et les réutiliser par la suite. Les réutiliser demande nettement moins de temps que les recalculer, c'est pourquoi l'algorithme termine les étapes rapidement. Les sections suivantes étudient plus en détail les tenants et les aboutissants de la programmation dynamique.

Trouver un fondement historique

La *programmation dynamique* consiste à faire en sorte qu'un algorithme garde en mémoire les résultats précédents afin d'éviter de devoir refaire plusieurs fois les mêmes calculs. Bien que cette notion puisse parfois sembler un peu compliquée, sa mise en application est en réalité très simple. Néanmoins, son histoire est intéressante.

Dans son autobiographie, *In the Eye of the Hurricane*, Bellman explique que ce terme a été retenu à la fois par nécessité et par commodité. Il écrit que ce choix était un moyen de dissimuler aux yeux de Charles Erwin Wilson, secrétaire d'État à la Défense sous la présidence d'Eisenhower, la nature véritable de ses travaux de recherche au sein de la RAND Corporation (une institution de recherche et développement financée par le gouvernement des États-Unis et par des financeurs privés). C'est ce qui lui aurait permis de conserver son emploi dans cette institution. Vous pouvez lire son explication plus en détail dans l'extrait publié sur la page Web suivante : http://smo.sogang.ac.kr/doc/dy_birth.pdf. Certains chercheurs contestent cette version concernant l'origine du terme, par exemple Stuart Russell et Peter Norvig, qui dans leur ouvrage *Artificial Intelligence : A Modern Approach*,

affirment qu'en réalité Bellman avait utilisé l'expression *programmation dynamique* dans un article datant de 1952, avant que Wilson devienne secrétaire d'État à la Défense en 1953 (et Wilson lui-même avait été P.-D.G de General Motors avant de devenir ingénieur dans la recherche et le développement).

Les langages de programmation informatique n'étaient pas encore très utilisés au temps où Bellman effectuait des travaux de recherche opérationnelle, une discipline consistant à appliquer les mathématiques à la recherche des meilleures décisions, principalement dans les problèmes de production et de logistique (mais qui est aussi utilisée pour résoudre d'autres types de problèmes pratiques). L'informatique n'en était encore qu'à ses débuts, et elle était utilisée principalement pour la planification. L'approche fondamentale de la programmation dynamique est la même que celle de la *programmation linéaire*, autre technique algorithmique (voir [Chapitre 19](#)) définie en un temps où programmation signifiait planification d'un processus particulier pour trouver une solution optimale. Le terme *dynamique* nous rappelle que l'algorithme agit par étapes et stocke des solutions partielles. Programmation dynamique est un nom compliqué pour désigner une technique subtile et efficace destinée à améliorer les temps d'exécution des algorithmes.

Rendre les problèmes dynamiques

Parce qu'elle tire parti des opérations répétées, la programmation dynamique est adaptée à la résolution des problèmes dont les solutions sont basées sur la résolution de sous-problèmes, l'algorithme assemblant ensuite les solutions partielles pour fournir une réponse complète. Pour fonctionner efficacement, la programmation dynamique résout des sous-problèmes imbriqués dans d'autres sous-problèmes (une méthode qui ressemble aux algorithmes gloutons, qui fonctionnent aussi avec une sous-structure optimale, comme expliqué au [Chapitre 15](#)). Pour faire mieux que les méthodes utilisant la force brute qui reviennent répétitivement sur les mêmes sous-problèmes, la programmation

dynamique doit être appliquée, elle aussi, à un problème décomposable en sous-problèmes.



La programmation dynamique est un concept englobant une grande diversité d'applications, et non pas un algorithme spécifique destiné à résoudre un problème particulier. Elle est plutôt une technique générale pour résoudre des problèmes.

La programmation dynamique se rattache à deux grandes familles de solutions :

- » **La méthode ascendante** : Consiste à produire une série de résultats partiels dont la réunion constitue la solution complète
- » **La méthode descendante** : Consiste à diviser le problème en sous-problèmes, en partant de la solution complète (une méthode typique des algorithmes récursifs) et en recourant à la mémorisation (définie dans la prochaine section) pour éviter de répéter plus d'une fois les calculs

L'approche descendante est généralement plus efficace du point de vue de l'informatique car elle ne génère que les sous-problèmes nécessaires à l'élaboration de la solution complète. L'approche ascendante a un caractère plus exploratoire, elle consiste à procéder par essais et erreurs et produit souvent des résultats qui ne serviront pas. D'un autre côté, les méthodes ascendantes reflètent mieux l'approche que l'on adopte généralement dans la vie quotidienne face à un problème (alors que la pensée récursive suppose un certain sens de l'abstraction et une formation pour pouvoir déboucher sur une application). Parfois, les deux approches peuvent sembler difficiles. La raison à cela est que la programmation dynamique change notre façon de résoudre les problèmes. Elle comporte les étapes suivantes :

1. **Création d'une solution pratique en utilisant la force brute ou la récursivité. La solution est viable, mais elle demande beaucoup de temps, quand il est possible de terminer la résolution du problème.**

2. **Stockage des résultats des sous-problèmes pour accélérer les calculs et aboutir à une solution dans un délai raisonnable.**
3. **Changement de la façon d'aborder le problème et gain supplémentaire en rapidité.**
4. **Redéfinition de la démarche de résolution du problème, sous une forme moins intuitive mais plus efficiente en vue de mieux exploiter la programmation dynamique.**

Quand la programmation dynamique transforme les algorithmes pour les rendre plus efficaces, ils deviennent plus difficiles à comprendre. On peut avoir l'impression que si les solutions fonctionnent, c'est comme par magie. Bien maîtriser la programmation dynamique suppose des observations répétées des solutions existantes et la résolution d'exercices pratiques. Cependant, cette maîtrise vaut la peine d'être acquise car la programmation dynamique permet de résoudre les problèmes pour lesquels il est indispensable de calculer et de comparer de façon systématique toutes les solutions possibles.



La programmation dynamique est surtout connue pour permettre de résoudre (ou du moins, de résoudre plus rapidement) des problèmes d'optimisation combinatoire, c'est-à-dire des problèmes dont la solution est une combinaison particulière d'éléments d'input. Les exemples classiques de ce type de problèmes sont notamment le problème du voyageur de commerce et le problème du sac à dos, qui sont présentés en détail plus loin dans ce chapitre.

Utiliser la récursivité dans un contexte dynamique

Le principe fondamental de la programmation dynamique est d'obtenir un résultat aussi exploitable que par la méthode de recherche par force brute, mais sans devoir consacrer tout son

temps à effectuer tous les calculs que celle-ci exige. Il s'agit d'échanger du temps contre de l'espace disque ou de l'espace mémoire, généralement en créant une structure de données (une table de hachage, un tableau ou une matrice) pour y stocker les résultats déjà obtenus. Cela permet d'accéder aux résultats sans devoir effectuer des calculs une seconde fois.



La technique consistant à stocker les résultats de la fonction précédente et à les utiliser plutôt que d'appeler cette fonction à nouveau s'appelle la *mémoïsation*, à ne pas confondre avec la mémorisation. Le terme *mémoïsation* vient du mot latin *memorandum* qui signifie « dont il faut se rappeler ».



La *mise en cache*, ou mise en antémémoire, est aussi un concept utilisé conjointement avec la notion de *mémoïsation*. Elle consiste à utiliser une zone spéciale de la mémoire de l'ordinateur pour délivrer des données plus rapidement dès qu'elles sont sollicitées. De façon générale, elle trouve plus d'applications que la *mémoïsation*.

La programmation dynamique est efficace pour résoudre des problèmes dont la résolution nécessite une répétition ou un rappel de certaines étapes. Un bon exemple de ce type de situation est le recours à la récursivité, comme dans le calcul des nombres de la suite de Fibonacci. La suite de Fibonacci est simplement une série de nombres dans laquelle chaque élément est la somme des deux nombres qui le précèdent. Le premier élément est 0, suivi de 1. Après avoir défini ces deux premiers éléments, il suffit à chaque fois d'additionner les deux derniers éléments pour obtenir le suivant. En voici les onze premiers :

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

Comme pour l'indexation avec Python, le comptage commence à la position zéro et le dernier nombre de la séquence est en dixième position. L'inventeur de cette suite, le mathématicien italien Leonardo Pisano, plus connu sous le nom de Fibonacci, a vécu au XII^e siècle. Il considérait que si chaque nombre était la somme des deux précédents, cette suite devait convenir pour représenter le

profil de croissance d'une population de lapins. Cette application à la démographie des lapins n'a pas été probante, mais la suite de Fibonacci a tout de même fait progresser les connaissances de manière fortuite, non seulement en mathématiques, mais aussi dans les sciences de la nature, car elle a été exploitée en botanique et en zoologie. On observe en effet cette progression dans le développement des branches des arbres, par exemple, ainsi que dans l'agencement des feuilles sur une tige et dans la disposition des graines sur une fleur de tournesol (voir <https://www.goldennumber.net/spirals/>).



Fibonacci est aussi le mathématicien qui a introduit en Europe les chiffres arabes, le système de notation que nous utilisons tous les jours. Il a décrit ces chiffres ainsi que sa suite dans son chef-d'œuvre le *Liber Abaci*, en 1202.

On peut calculer une séquence de nombres de la suite de Fibonacci en utilisant la récursivité. Quand on entre un nombre, la fonction de récursivité le remplace par les deux nombres qui le précèdent dans la suite de Fibonacci. Ensuite, elle effectue la même tâche pour chaque élément obtenu, et remplace chacun des deux nombres ainsi calculés par les deux nombres qui le précèdent, et ainsi de suite jusqu'à la racine de la suite, constituée des nombres 0 et 1. Des deux types de programmation dynamique évoqués au paragraphe précédent, c'est la méthode descendante qui est utilisée ici. Le code suivant est une application de la méthode récursive avec Python (vous pouvez retrouver ce code dans le fichier téléchargeable A4D ; 16 ; Fibonacci.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
def fib(n, tab=0):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        print ("lvl %i, Addition de fib(%i) et fib(%i)"
%
        (tab, n-1, n-2))
        return fib(n-1,tab+1) + fib(n-2,tab+1)
```

Le programme affiche la décomposition produite par chaque niveau de récursivité. L'output suivant montre ce qui se produit quand la fonction fib() est appelée avec une valeur d'entrée de 7 :

```
fib(7)

lvl 0, Addition de fib(6) et fib(5)
lvl 1, Addition de fib(5) et fib(4)
lvl 2, Addition de fib(4) et fib(3)
lvl 3, Addition de fib(3) et fib(2)
lvl 4, Addition de fib(2) et fib(1)
lvl 5, Addition de fib(1) et fib(0)
lvl 4, Addition de fib(1) et fib(0)
lvl 3, Addition de fib(2) et fib(1)
lvl 4, Addition de fib(1) et fib(0)
lvl 2, Addition de fib(3) et fib(2)
lvl 3, Addition de fib(2) et fib(1)
lvl 4, Addition de fib(1) et fib(0)
lvl 3, Addition de fib(1) et fib(0)
lvl 1, Addition de fib(4) et fib(3)
lvl 2, Addition de fib(3) et fib(2)
lvl 3, Addition de fib(2) et fib(1)
lvl 4, Addition de fib(1) et fib(0)
lvl 3, Addition de fib(1) et fib(0)
lvl 2, Addition de fib(2) et fib(1)
lvl 3, Addition de fib(1) et fib(0)
```

13

L'output comporte 20 décompositions. Certains nombres y apparaissent plus d'une fois. Cet exemple semble idéal pour appliquer la programmation dynamique.

Le code suivant ajoute un dictionnaire, appelé memo, qui stocke les résultats précédemment obtenus. Après avoir décomposé un nombre, la fonction de récursivité vérifie si le résultat apparaît déjà dans le dictionnaire avant d'entamer la prochaine branche récursive. Lorsqu'il trouve le résultat, le programme utilise le résultat déjà calculé :

```
memo = dict()
def fib_mem(n, tab=0):
    if n==0:
        return 0
```

```

elif n == 1:
    return 1
else:
    if (n-1, n-2) not in memo:
        print ("lvl %i, Addition de fib(%i) et
fib(%i)" %
            (tab, n-1, n-2))
        memo[(n-1,n-2)] = fib_mem(n-1,tab+1
            ) + fib_mem(n-2,tab+1)
    return memo[(n-1,n-2)]

```

Grâce à la mémoïsation, la fonction récursive calcule non pas 20 additions, mais seulement six, celles qui sont essentielles pour résoudre le problème initial consistant à calculer un certain nombre de la série :

```

fib_mem(7)

lvl 0, Addition de fib(6) et fib(5)
lvl 1, Addition de fib(5) et fib(4)
lvl 2, Addition de fib(4) et fib(3)
lvl 3, Addition de fib(3) et fib(2)
lvl 4, Addition de fib(2) et fib(1)
lvl 5, Addition de fib(1) et fib(0)

```

13

Dans le dictionnaire memo, on trouve la succession de sommes qui définit la suite de Fibonacci à partir de 1 :

```

memo
{(1, 0): 1, (2, 1): 2, (3, 2): 3, (4, 3): 5, (5, 4):
8,
(6, 5): 13}

```

Exploiter la mémoïsation

La mémoïsation est le principe même de la programmation dynamique. Souvent, dans l'écriture d'un algorithme, on se retrouve obligé d'y recourir. Quand vous créez une fonction, qu'elle soit ou non récursive, vous pouvez facilement la transformer à l'aide d'une commande simple, un *décorateur*. Il

s'agit d'une fonction Python spéciale qui transforme les fonctions. Pour savoir comment l'utiliser, commencez par une fonction récursive, dépouillée de toute instruction print :

```
def fib(n):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Quand vous utilisez Jupyter, vous employez des commandes magiques intégrées d'IPython comme `timeit` pour mesurer le temps d'exécution d'une commande sur votre ordinateur :

```
%timeit -n 1 -r 1 print(fib(36))

14930352
1 loop, best of 1: 15.5 s per loop
```

Cet output montre que le temps d'exécution de la fonction est d'environ 15 secondes. Cependant, selon la machine que vous utilisez, ce temps d'exécution peut être plus ou moins long. Quelle que soit la rapidité de votre ordinateur, il lui faudra certainement plusieurs secondes pour terminer la tâche, car le 36^e nombre de Fibonacci est très grand : 14 930 352. Tester la même fonction pour des nombres de la suite de Fibonacci plus élevés demandera plus de temps encore.

Il est temps à présent d'observer l'effet de la décoration de la fonction. La fonction `lru_cache` du module `functools` permet de réduire considérablement le temps d'exécution. Cette fonction n'est disponible qu'avec Python 3. Elle transforme une fonction en ajoutant automatiquement un cache pour conserver ses résultats. Vous pouvez aussi paramétrer la taille du cache à l'aide du paramètre `maxsize` . Avec l'instruction `maxsize=None`, le cache utilise toute la mémoire disponible, sans limites.

```
from functools import lru_cache
```



```
@lru_cache(maxsize=None)
def fib(n):
    if n==0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Il convient de noter que la fonction est la même que précédemment. Le seul ajout est la fonction importée `lru_cache` (<https://docs.python.org/3.5/library/functools.html>), appelée en plaçant devant son nom le symbole `@`. Le symbole `@` (arobase) est une annotation pour appeler la fonction `lru_cache` comme décorateur de la fonction qui suit.



L'utilisation des décorateurs est une technique avancée de Python. Il n'est pas utile d'expliquer en détail les décorateurs dans ce livre, mais vous pouvez toujours en profiter car ils sont très faciles à utiliser (pour plus de renseignements sur les décorateurs, consultez les pages <http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/> et <https://www.learnpython.org/en/Decorators>). Simplement, n'oubliez pas que pour appeler un décorateur il faut utiliser une annotation (`@` + nom de la fonction décorateur), et qu'il faut placer le décorateur avant la fonction à transformer. La fonction originale est transmise au décorateur et revient transformée. Dans cet exemple de fonction récursive simple, le décorateur donne une fonction de récursivité enrichie par la mémoïsation.

Il est temps de tester la rapidité de la fonction, comme précédemment :

```
%timeit -n 1 -r 1 print(fib(36))

14930352
1 loop, best of 1: 60.6 _s per loop
```

Même si vous obtenez un temps d'exécution différent, ce temps ne doit plus être de l'ordre de quelques secondes, mais de

quelques millisecondes. Tel est le pouvoir de la mémorisation. Vous pouvez examiner la façon dont votre fonction utilise son cache en appelant la fonction `cache_info` à partir de la fonction décorée :

```
fib.cache_info()

CacheInfo(hits=34, misses=37, maxsize=None,
currsize=37)
```

Cet output indique que 37 appels de fonctions ne trouvent pas de réponse dans le cache. Cependant, 34 autres appels y ont trouvé une réponse utile.



En important simplement `lru_cache` à partir de `functools` et en l'utilisant pour les annotations devant vos principaux algorithmes dans Python, vous obtiendrez une nette augmentation des performances (sauf s'il s'agit d'algorithmes gloutons).

Découvrir les meilleures formules dynamiques

Même la programmation dynamique a ses limites. La plus importante de toutes est liée à son principal avantage : en gardant en mémoire trop de solutions partielles par souci de réduire le temps d'exécution, on risque de saturer la mémoire. Il peut y avoir trop de solutions partielles en mémoire lorsque le problème est complexe, ou simplement lorsque l'instruction utilisée pour produire les solutions partielles n'est pas optimale et lorsqu'un trop grand nombre de solutions partielles ne satisfont pas aux exigences à respecter.

L'instruction utilisée pour résoudre les sous-problèmes mérite votre attention. Il faut qu'elle soit adaptée à la progression efficace de l'algorithme (vous aboutissez à un résultat que vous allez réutiliser immédiatement), car tout repose sur la réutilisation intelligente des éléments précédemment assemblés. Par

conséquent, le recours à la mémorisation peut ne pas être suffisamment avantageux. Pour améliorer les résultats, réorganisez les problèmes de la meilleure manière. Vous pouvez apprendre à le faire en vous inspirant des meilleurs exemples de programmation dynamique, qui sont étudiés dans les sections suivantes : le sac à dos, le voyageur de commerce et la recherche approximative de chaîne.

Quand l'affaire est dans le sac

Le problème du sac à dos est connu au moins depuis 1897, et sa formalisation est probablement due à Tobias Dantzig (<https://www.britannica.com/biography/Tobias-Dantzig>). Ce problème consiste à ranger dans un sac à dos le plus d'objets possible. Chaque objet ayant une valeur, il s'agit de maximiser la valeur totale du chargement. Le sac à dos a une capacité de chargement donnée, ou bien c'est vous qui avez comme contrainte une limite de poids à transporter, si bien que vous ne pouvez pas tout emporter.

De façon générale, cette situation est celle de tout problème centré sur un budget et des ressources que l'on désire allouer de la façon la plus judicieuse possible. Ce type de problème est si courant que l'exemple du sac à dos est souvent considéré comme un problème d'algorithmique type. Il trouve des applications dans des domaines comme l'informatique, l'industrie, la finance, la logistique et la cryptographie. Comme applications concrètes du problème du sac à dos, on peut citer le problème du chargement optimal d'un cargo, ou la découpe optimale de matières premières pour qu'il y ait le moins de déchets possible.



Bien que le problème du sac à dos soit très connu, ce livre n'en propose pas à nouveau une étude détaillée, sachant que la méthode dynamique est incontestablement une des meilleures méthodes de résolution de problème. Il est important cependant de ne pas oublier que dans certains cas, notamment quand les éléments sont des quantités, d'autres méthodes, comme les algorithmes

gloutons, peuvent convenir aussi bien (voire mieux).

Cette section montre comment résoudre le *problème du sac à dos 1-0*. On dispose d'un nombre fini d'éléments qui peuvent être rangés dans le sac à dos (état 1) ou ne pas y être (état 0). Il importe de noter qu'il existe des variantes :

- » **Le problème du sac à dos à contenu fractionné :** Les éléments sont des quantités, par exemple 3 kg de farine, 1 kg de farine, 4 kg de farine, *etc.* Il s'agit de choisir les quantités les plus appropriées. Cette variante peut être résolue en utilisant un algorithme glouton.
- » **Le problème du sac à dos non extensible :** Mettre dans le sac un ou plusieurs exemplaires du même objet. Les contraintes sont exprimées en termes de nombre minimum et de nombre maximum pour chaque objet à sélectionner.
- » **Le problème du sac à dos extensible :** Mettre dans le sac un ou plusieurs exemplaires du même objet, sans contrainte. La seule restriction est qu'il n'est pas possible d'y mettre un nombre négatif d'éléments.

Le problème du sac à dos 1-0 se résout par la programmation dynamique, en temps pseudo-polynomial (ce qui est pire que le temps polynomial) sachant que le temps d'exécution dépend du nombre d'éléments (n) multiplié par le nombre de fractions de la capacité du sac (W) qui sont utilisées pour aboutir à la solution partielle. Avec la notation Big-O, on peut dire que le temps d'exécution est $O(nW)$. En revanche, la version force brute de l'algorithme s'exécute dans un temps $O(2^n)$. L'algorithme fonctionne de la façon suivante :

1. **Compte tenu de la capacité du sac à dos, tester une série de sacs plus petits (sous-problèmes). En l'occurrence, étant donné un sac capable de supporter 20 kilos, l'algorithme teste une série de sacs dont les capacités varient de 0 à 20 kilos.**

2. Pour chaque élément, tester la possibilité de le ranger dans chaque sac, du plus petit au plus grand. À chaque test, si l'élément entre dans le sac, choisir la meilleure valeur parmi les suivantes :

- a. la solution que représente le dernier sac plus petit ;
- b. l'élément testé, en remplissant l'espace qui reste avec la meilleure solution de remplissage d'un tel espace parmi les résultats qui précèdent.

Le code suivant exécute l'algorithme du sac à dos et résout le problème avec une série de six éléments représentant des combinaisons différentes de poids et de valeur et un sac de 20 kg :

Élément	1	2	3	4	5	6
Poids en kg	2	3	4	4	5	9
Profit en centaines d'euros	3	4	3	5	8	10

Voici le code pour exécuter la procédure de programmation dynamique décrite (vous le retrouverez dans le fichier téléchargeable A4D ; 16 ; Knapsack.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import numpy as np

values = np.array([3,4,3,5,8,10])
weights = np.array([2,3,4,4,5,9])
items = len(weights)
capacity = 20

memo = dict()
for size in range(0, capacity+1, 1):
    memo[(-1, size)] = ([], 0)

for item in range(items):
    for size in range(0, capacity+1, 1):
        # if the object doesn't fit in the knapsack
        if weights[item] > size:
            memo[item, size] = memo[item-1, size]
        else:
```

```

        # if the object fits, we check what can best
fit
        # in the residual space
previous_row, previous_row_value = memo[
    item-1, size-weights[item]]
if memo[item-1, size][1] > values[item
    ] + previous_row_value:
    memo[item, size] = memo[item-1, size]
else:
    memo[item, size] = (previous_row + [item
        ], previous_row_value + values[item])

```

La meilleure solution est le résultat qui est en cache lorsque le code teste l'ajout du dernier élément dans le sac de capacité maximale (20 kg) :

```

best_set, score = memo[items-1, capacity]
print ('La meilleure combinaison %s pèse %i et vaut
%i'
      % (best_set, np.sum((weights[best_set])), score))

```

La meilleure combinaison [0, 3, 4, 5] pèse 20 et vaut 26

Peut-être êtes-vous curieux de savoir ce qui s'est passé à l'intérieur du dictionnaire de mémorisation :

```

print (len(memo))

147

print (memo[2, 10])

([0, 1, 2], 10)

```

Il contient 147 sous-problèmes. Six éléments multipliés par 21 sacs à dos donnent 126 solutions, mais il faut ajouter encore 21 solutions naïves pour que l'algorithme fonctionne correctement (*naïve* signifie laissant le sac à dos vide), ce qui porte le nombre de sous-problèmes à 147.

La tâche de résoudre 147 sous-problèmes peut vous sembler rebutante (même si leur résolution est très rapide). Dans ce cas

particulier, le recours à la seule force brute implique la résolution d'un nombre de sous-problèmes moins important, et donc un temps de traitement moins long, ce que vous pouvez vérifier en utilisant Python et la fonction `comb` :

```
from scipy.misc import comb
objects = 6
np.sum([comb(objects,k+1) for k in range(objects)])
```

Il faut tester 63 combinaisons pour résoudre ce problème. Cependant, si vous essayez d'utiliser davantage d'objets, par exemple 20 objets, les temps d'exécution seront très différents car il y aura alors 1 048 575 combinaisons à tester. Comparez ce nombre colossal avec la programmation dynamique, avec laquelle il n'y a que $20 * 21 + 21 = 441$ sous-problèmes à résoudre.



C'est là la différence entre le temps quasi-polynomial et le temps exponentiel (pour mémoire, ce livre étudie la complexité exponentielle au [Chapitre 2](#), à propos de la notation Big-O, et le [Chapitre 15](#) parle du temps polynomial dans le cadre d'une étude portant sur les problèmes NP-complets). La programmation dynamique est bien utile quand les problèmes à résoudre sont complexes. Les problèmes simplifiés sont utiles pour l'apprentissage, mais ils ne peuvent pas bien refléter l'utilisation de techniques algorithmiques intelligentes comme la programmation dynamique. Chaque solution teste le résultat de l'ajout d'un certain élément quand le sac à dos a une capacité donnée. Dans l'exemple précédent, on ajoute l'élément 2 (poids = 4, valeur = 3) et l'on obtient une solution consistant à placer les éléments 0, 1 et 2 dans le sac (poids total 9 kg) pour une valeur de 10. Cette solution intermédiaire tire parti des solutions précédentes et sert de base à un certain nombre de solutions ultérieures avant que l'algorithme ne termine le traitement.



Vous vous demandez peut-être si le résultat produit par ce script est vraiment le meilleur que l'on puisse obtenir. Malheureusement, la seule façon d'en être sûr est de connaître la bonne réponse, ce qui implique l'utilisation d'un algorithme de force brute (lorsque c'est envisageable en termes de temps

d'exécution sur l'ordinateur). Ce chapitre ne traite pas de l'utilisation de la force brute concernant le problème du sac à dos, mais l'approche par la force brute est utilisée dans l'exemple qui suit, celui du voyageur de commerce.

Effectuer une tournée des villes

Le problème du voyageur de commerce est au moins aussi connu que le problème du sac à dos. Il est utilisé essentiellement dans les domaines de la logistique et des transports (comme le problème des tournées de véhicules qui en dérive, et qui est présenté sur la page <http://neo.lcc.uma.es/vrp/vehicle-routing-problem/>). Son champ d'application est donc plus limité que celui du problème du sac à dos. Le problème est le suivant : un voyageur de commerce doit visiter un certain nombre de villes, puis regagner la ville qui constitue son point de départ (c'est donc un voyage circulaire, c'est pourquoi on parle de tournée) en parcourant la moins grande distance possible.

Ce problème est similaire aux problèmes d'optimisation à base de graphe, mais sans les arêtes car les villes sont toutes interconnectées. C'est pourquoi on utilise une matrice des distances comme input, c'est-à-dire un tableau dans lequel les villes apparaissent en ligne et en colonne. Chaque intersection contient la distance entre la ville correspondant à la ligne courante et la ville correspondant à la colonne courante. Il existe des variantes de ce problème, dans lesquelles la matrice contient les temps de trajet ou les consommations de carburant, et non les distances.

Le problème du voyageur de commerce est un problème de type NP, mais il peut être résolu par plusieurs méthodes, certaines approximatives (méthodes heuristiques), certaines précises (programmation dynamique). Le problème, comme pour tout autre problème de type NP, est le temps d'exécution. On peut trouver des solutions que l'on supposera optimales : on peut s'en assurer quand il s'agit de tournées réduites, mais pas quand il s'agit de

problèmes aussi complexes que parcourir le monde : <http://www.math.uwaterloo.ca/tsp/world/>. Dans l'exemple suivant, on essaie divers algorithmes (force brute, algorithme glouton et programmation dynamique) sur une tournée simple comportant six villes. Le problème est représenté par un graphe pondéré (voir Figure 16-1) (vous le retrouverez dans le fichier téléchargeable A4D ; 16 ; TSP.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

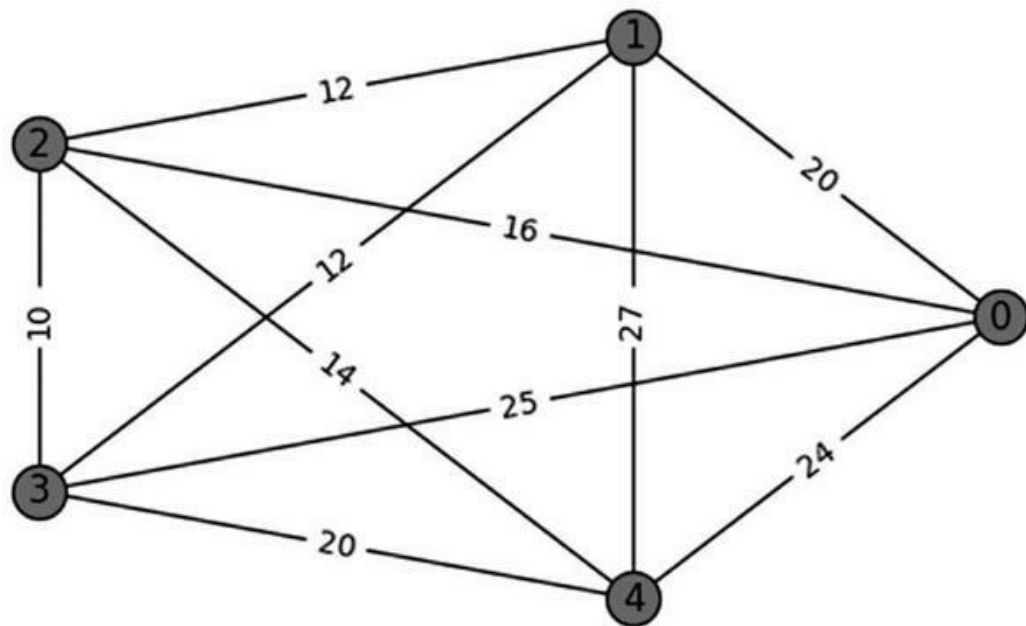


FIGURE 16-1 Les villes sont représentées par des sommets sur un graphe pondéré.

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

D = np.array([[0, 20, 16, 25, 24], [20, 0, 12, 12, 27],
              [16, 12, 0, 10, 14], [25, 12, 10, 0, 20],
              [24, 27, 14, 20, 0]])

Graph = nx.Graph()
Graph.add_nodes_from(range(D.shape[0]))
for i in range(D.shape[0]):
    for j in range(D.shape[0]):
```

```

Graph.add_edge(i,j,weight=D[i,j])

np.random.seed(2)
pos=nx.shell_layout(Graph)
nx.draw(Graph, pos, with_labels=True)
labels = nx.get_edge_attributes(Graph, 'weight')
nx.draw_networkx_edge_labels(Graph,pos,
                             edge_labels=labels)
plt.show()

```

Après avoir défini la matrice D (distance), le programme essaie la première solution, la plus simple, pour déterminer la tournée la plus courte commençant et finissant à la ville numéro zéro. Cette solution procède de la méthode de la force brute, qui produit toutes les permutations possibles en laissant de côté le point zéro. Les distances entre la ville numéro zéro et la première ville et entre la dernière ville de la tournée et la ville numéro zéro sont ajoutées après avoir calculé la distance totale dans chaque solution. Quand toutes les solutions sont envisageables, on choisit simplement la plus courte.

```

from itertools import permutations
best_solution = [None, np.sum(D)]
for solution in
list(permutations(range(1,D.shape[0]))):
    start, distance = (0,0)
    for next_one in solution:
        distance += D[start, next_one]
        start = next_one
    distance += D[start,0]
    if distance <= best_solution[1]:
        best_solution = [[0]+list(solution)+[0],
distance]
        print ('Meilleure solution jusqu'ici : %s kms' %
                str(best_solution)[1:-1])

```

```

Meilleure solution jusqu'ici : [0, 1, 2, 3, 4, 0], 86
kms
Meilleure solution jusqu'ici : [0, 1, 3, 2, 4, 0], 80
kms
Meilleure solution jusqu'ici : [0, 4, 2, 3, 1, 0], 80
kms

```

L'algorithme de force brute détermine rapidement la meilleure solution et le chemin qui lui est symétrique. Cependant, si l'on obtient rapidement un résultat, c'est parce qu'avec quatre villes, il n'existe que 24 solutions possibles. À mesure que le nombre de villes augmente, le nombre de permutations à tester devient inextricable, même après avoir éliminé les trajets symétriques (ce qui divise par deux le nombre de permutations) et même en utilisant un ordinateur performant. Considérons, par exemple, le nombre de calculs à faire s'il y a 13 villes sans compter le point de départ et d'arrivée :

```
from scipy.special import perm
print (perm(13,13)/2)

3113510400.0
```

La programmation dynamique peut simplifier le temps d'exécution. L'algorithme de Held-Karp (aussi appelé algorithme de Bellman-Held-Karp, car Bellman l'avait publié en 1962, la même année que Michael Held et Richard Karp) peut réduire la complexité en temps à $O(2^n n^2)$. C'est toujours une complexité exponentielle, mais avec un temps d'exécution plus court que celui qu'exige l'énumération exhaustive de toutes les tournées dans le cas de la force brute.



Les algorithmes des méthodes approximative et heuristique peuvent produire rapidement des résultats exploitables (même si le résultat ne reflète pas toujours la solution optimale, il est généralement assez satisfaisant). Vous retrouverez le problème du voyageur de commerce plus loin dans ce livre (voir Chapitres [18](#) et [20](#)), à propos de la recherche locale et de l'heuristique.

Afin de trouver la meilleure solution au problème du voyageur de commerce pour n villes, en commençant et en finissant par la ville 0, l'algorithme part de la ville 0 et garde en mémoire le plus court chemin possible en prenant en compte différentes possibilités. Il utilise toujours une ville finale différente en ne couvrant qu'un sous-ensemble des villes. À mesure que ce sous-

ensemble s'agrandit, l'algorithme apprend à résoudre le problème efficacement. Lorsqu'il doit résoudre le problème pour cinq villes, l'algorithme prend donc d'abord en compte les solutions qui concernent deux villes, puis trois villes, puis quatre, et enfin cinq (la dimension du sous-ensemble variant de 1 à n). L'algorithme passe par les étapes suivantes :

- 1. Initialisation d'un tableau des distances de la ville 0 à toutes les autres villes. À cette étape initiale, les trajets envisagés se font simplement entre la ville de départ et une destination.**
- 2. Prise en compte de toutes les dimensions possibles du sous-ensemble de villes, depuis deux jusqu'au nombre total de villes concernées par la tournée. Il s'agit de la première itération, de la boucle extérieure.**
- 3. À l'intérieur de la boucle extérieure, pour chaque dimension du sous-ensemble de villes, prise en compte de toutes les combinaisons possibles de ces villes, la ville initiale n'étant pas incluse. Il s'agit d'une itération intérieure.**
- 4. À l'intérieur de l'itération intérieure (étape 3), pour toute combinaison possible, prise en compte de chaque ville à l'intérieur de la combinaison comme ville finale. Il s'agit d'une autre itération intérieure.**
- 5. À l'intérieur de l'itération intérieure (étape 4), pour une ville de destination différente, détermination du plus court chemin reliant les villes du sous-ensemble considéré à la ville de départ de la tournée (ville 0). Dans la détermination du plus court chemin, utilisation de toutes les informations utiles précédemment stockées (application de la programmation dynamique). Cette étape évite des calculs et justifie la progression par sous-ensembles de villes de plus en plus grands. L'exploitation des solutions précédemment trouvées aux sous-problèmes permet de trouver les trajectoires les moins**

longues en ajoutant au plus court chemin trouvé à l'étape précédente la distance nécessaire pour atteindre la ville de destination. Pour un certain sous-ensemble de villes et une ville initiale donnée, l'algorithme stocke le meilleur chemin et sa longueur.

- 6. Quand toutes les itérations sont terminées, on obtient n-1 solutions différentes, en termes de plus court chemin, chaque solution couvrant toutes les villes mais comportant une ville finale différente. Ajout d'un point de fermeture, la ville 0, à chaque solution pour conclure la tournée.**
- 7. Détermination de la solution la plus courte et sélection de cette solution comme résultat.**



L'implémentation de cet algorithme dans Python est un peu compliquée en raison des itérations et de la manipulation de sous-ensembles. Il s'agit d'une recherche exhaustive renforcée par la programmation dynamique et utilisant une approche itérative avec des sous-ensembles de villes et des villes candidates à leur ajouter. L'exemple commenté suivant, avec Python, montre comment fonctionne cette méthode. Vous pouvez personnaliser cet exercice de détermination de tournées (en utilisant des villes de votre région ou de votre département comme entrées dans la matrice des distances). Pour aboutir à la solution, le script utilise des instructions avancées comme frozenset (une instruction qui génère un sous-ensemble utilisable comme clé de dictionnaire) et des opérateurs pour les sous-ensembles.

```
from itertools import combinations
memo = {(frozenset([0, idx+1]), idx+1): (dist,
[0,idx+1])

    for idx,dist in enumerate(D[0][1:])}
cities = D.shape[0]
for subset_size in range(2, cities):
    # Ici on définit la taille du sous-ensemble de
    villes
    new_memo = dict()
    for subset in [frozenset(comb) | {0} for comb in
```

```

        combinations(range(1, cities),
            subset_size)]:
    # On énumère les sous-ensembles ayant une taille
    # donnée
    for ending in subset - {0}:
        # Prise en compte de chaque point d'arrivée dans
le sous-en-
semble
        all_paths = list()
        for k in subset:
            # Détermination du plus court chemin pour
tout
            # élément du sous-ensemble
            if k != 0 and k!=ending:
                length = memo[(subset-{ending},k)][0]
                + D[k][ending]
                index = memo[(subset-{ending},k)][1]
                + [ending]
                all_paths.append((length, index))
            new_memo[(subset, ending)] = min(all_paths)
        # Pour économiser de la mémoire, on enregistre
seulement les
        # sous-ensembles précédents car les plus petits ne
seront plus uti-
lisés
        memo = new_memo
    # Fin du cycle et retour au début de la tournée,
    # à la ville numéro zéro
    tours = list()
    for distance, path in memo.values():
        distance += D[path[-1],0]
        tours.append((distance, path+[0]))
    # On peut maintenant annoncer la trajectoire la plus
courte
    distance, path = min(tours)
    print ('Solution la plus courte par la programmation
dynamique : %s,
%i kms'
        % (path, distance))

Solution la plus courte par la programmation dynamique
:
[0, 1, 3, 2, 4, 0], 80 kms

```

La recherche par approximation

dans les chaînes

Il n'est pas toujours simple de déterminer si un mot est similaire à un autre. Des mots peuvent différer légèrement par suite d'une faute d'orthographe ou parce qu'un mot peut s'écrire de plus d'une manière, avec pour conséquence qu'une correspondance exacte devient impossible. Il ne s'agit cependant pas simplement de questions d'orthographe, aussi intéressantes soient-elles. À titre d'exemple, rapprocher des chaînes de caractères similaires (notamment des noms, des adresses, ou des codes d'identification) faisant référence à la même personne peut permettre d'obtenir une vue client unique dans la base de clientèle d'une entreprise, ou bien, au sein de services de sécurité, de localiser un dangereux criminel.



La recherche par approximation connaît des applications variées dans des domaines comme la traduction automatique, la reconnaissance vocale, la correction orthographique et le traitement de texte, la biologie computationnelle et la récupération d'informations. Concernant la façon dont les sources de données alimentent les bases de données, il existe de nombreuses disparités entre les champs de données qu'un algorithme intelligent doit traiter. La capacité d'établir une correspondance entre deux séries de caractères similaires, mais pas précisément égales, trouve des applications dans des domaines comme la génétique, lorsqu'il s'agit de comparer deux séquences d'ADN (exprimées par les lettres qui représentent les nucléotides, G,A,T, et C) en vue de déterminer si elles sont similaires et d'établir leur degré de ressemblance.



Vladimir Levenshtein, scientifique russe spécialisé dans la théorie de l'information (pour plus de détails, voir http://ethw.org/Vladimir_I_Levenshtein), a conçu en 1965 une simple mesure (qui porte son nom) du degré de similarité entre deux chaînes de caractères, consistant à compter le nombre de transformations nécessaires pour passer d'une chaîne à l'autre. La distance de Levenshtein (aussi appelée distance d'édition) est le nombre de changements de caractères dans un mot :

- » **Suppression** : Suppression d'une lettre dans un mot.
- » **Insertion** : Ajout d'une lettre dans un mot pour obtenir un autre mot.
- » **Substitution** : Remplacement d'une lettre par une autre, par exemple de la lettre *m* par la lettre *p* pour obtenir *pot* à partir de *mot*.



Chaque changement a un coût, que Levenshtein fixe à 1 unité pour chaque transformation. Cependant, selon la façon dont on applique l'algorithme, on peut établir un coût différent pour la suppression, l'insertion et la substitution. Quand on recherche des noms de rues similaires, par exemple, les noms mal orthographiés sont plus courants que les différences véritables, par conséquent on peut attribuer à la substitution un coût de 1, et à la suppression ainsi qu'à l'insertion un coût de 2. D'un autre côté, lorsqu'il s'agit de montants monétaires, des valeurs similaires peuvent très bien présenter un nombre de chiffres différent, par exemple si l'on peut saisir aussi bien 123 € que 123,00 €. Par conséquent, on peut défendre l'idée que l'insertion et la suppression coûtent moins cher que la substitution (124 € n'est pas exactement la même valeur que 123 €, par conséquent substituer un 3 à un 4 devrait coûter plus cher).

L'algorithme de comptage peut être récursif ou itératif. Cependant, il fonctionne bien plus rapidement lorsqu'il est conçu selon une programmation dynamique ascendante, comme l'explique un article de Robert A. Wagner et Michael J. Fischer paru en 1974 et intitulé « The String-to-string Correction Problem »

(<http://www.inrg.csie.ntu.edu.tw/algorithm2014/homework/Wagner74.pdf>). La complexité en temps de cette solution est $O(mn)$, où n et m sont les longueurs en nombre de lettres des deux mots à comparer. Le code suivant calcule le nombre de changements nécessaires pour transformer le mot Saturday en Sunday en utilisant la programmation dynamique, avec une matrice ([voir Figure 16-2](#)) pour stocker les résultats intermédiaires (approche ascendante) (vous le retrouverez dans le fichier téléchargeable

A4D ; 16 ; Levenshtein.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

		s	u	n	d	a	y
	0.0	1.0	2.0	3.0	4.0	5.0	6.0
s	1.0	0.0	1.0	2.0	3.0	4.0	5.0
a	2.0	1.0	1.0	2.0	3.0	3.0	4.0
t	3.0	2.0	2.0	2.0	3.0	4.0	4.0
u	4.0	3.0	2.0	3.0	3.0	4.0	5.0
r	5.0	4.0	3.0	3.0	4.0	4.0	5.0
d	6.0	5.0	4.0	4.0	3.0	4.0	5.0
a	7.0	6.0	5.0	5.0	4.0	3.0	4.0
y	8.0	7.0	6.0	6.0	5.0	4.0	3.0

FIGURE 16-2 Transformer Sunday en Saturday.

```
import numpy as np
import pandas as pd

s1 = 'Saturday'
s2 = 'Sunday'
m = len(s1)
n = len(s2)
D = np.zeros((m+1,n+1))
D[0,:] = list(range(n+1))
D[:,0] = list(range(m+1))

for j in range(1, n+1):
    for i in range(1, m+1):
        if s1[i-1] == s2[j-1]:
            D[i, j] = D[i-1, j-1]
        else:
            D[i, j] = np.min([
```

```

        D[i-1, j] + 1, # une suppression
        D[i, j-1] + 1, # une insertion
        D[i-1, j-1] + 1 # une substitution
    ])
print ('La distance de Levenshtein est égale à %i' %
      D[-1, -1])

```

La distance de Levenshtein est égale à 3

Vous pouvez faire apparaître le résultat à l'aide de l'instruction suivante :

```

pd.DataFrame(D, index=list(' '+s1), columns=list(' '+s2))

```

L'algorithme construit la matrice et inscrit la meilleure solution dans la dernière cellule. Après avoir constitué la matrice en utilisant les lettres de la première chaîne pour les rangées et les lettres de la seconde pour les colonnes, le programme procède par colonnes et évalue la différence entre les lettres des rangées et celles des colonnes. Ainsi, l'algorithme effectue un nombre de comparaisons équivalent au produit des nombres de lettres des deux chaînes. Il prend en compte le résultat des comparaisons précédentes en examinant les solutions présentes dans les cellules précédentes de la matrice et en choisissant la solution comportant le plus petit nombre de modifications.

Quand l'itération matricielle est terminée, le résultat est le nombre minimum de modifications nécessaires à la transformation : plus ce nombre est petit, plus les deux chaînes sont similaires. Le retour de la dernière cellule vers la première en gagnant à chaque fois la cellule précédente dont la valeur est la plus petite (quand il y a plus d'une direction possible, le programme privilégie le déplacement en diagonale) indique les transformations à exécuter ([voir Figure 16-3](#)) :

- » Un mouvement arrière en diagonale correspond à une substitution dans la première chaîne si les lettres de la ligne et de la colonne diffèrent (sans quoi aucune modification n'est nécessaire).

- » Un mouvement vers le haut indique la suppression d'une lettre dans la première chaîne.
- » Un mouvement en arrière vers la gauche indique qu'une nouvelle lettre doit être supprimée dans la première chaîne de caractères.

Dans cet exemple, le retour en arrière fait apparaître les transformations suivantes (deux suppressions et une substitution) :

Saturday => Sturday => Surday => Sunday

		s	u	n	d	a	y
	0.0	1.0	2.0	3.0	4.0	5.0	6.0
s	1.0	<u>0.0</u>	1.0	2.0	3.0	4.0	5.0
a	2.0	<u>1.0</u>	1.0	2.0	3.0	3.0	4.0
t	3.0	<u>2.0</u>	2.0	2.0	3.0	4.0	4.0
u	4.0	3.0	2.0	3.0	3.0	4.0	5.0
r	5.0	4.0	3.0	<u>3.0</u>	4.0	4.0	5.0
d	6.0	5.0	4.0	4.0	3.0	4.0	5.0
a	7.0	6.0	5.0	5.0	4.0	3.0	4.0
y	8.0	7.0	6.0	6.0	5.0	4.0	3.0

FIGURE 16-3 Montrer quelles transformations ont lieu.

Chapitre 17

Utiliser des algorithmes probabilistes

DANS CE CHAPITRE

- » Comprendre comment une démarche aléatoire peut se révéler plus judicieuse qu'une méthode plus rationnelle
 - » Formuler des idées clés sur la probabilité et ses distributions
 - » Étudier le fonctionnement d'une simulation de Monte Carlo
 - » Découvrir Quickselect et réviser les algorithmes Quicksort
-

Les générateurs de nombres au hasard ont une fonction essentielle en informatique et jouent un rôle important dans les techniques algorithmiques étudiées dans cette partie du livre. La randomisation n'est pas utilisée uniquement dans le domaine des jeux et des paris. Elle sert à résoudre une grande variété de problèmes. Elle se révèle parfois plus efficace que d'autres techniques pour l'optimisation et pour l'obtention de la bonne solution. Elle permet une meilleure utilisation de certaines autres techniques, qu'il s'agisse de la recherche locale, du recuit simulé, de la méthode heuristique, de la cryptographie ou de l'informatique distribuée (la cryptographie étant la technique la plus importante pour dissimuler l'information).

On retrouve la randomisation dans des outils de tous les jours, là où on ne l'attend pas toujours. Le robot aspirateur Roomba (mis au point par une société fondée par le Massachusetts Institute of

Technology, le MIT) circule sur le sol d'une pièce sans trajectoire planifiée et sans utiliser un plan des lieux à parcourir. La plupart du temps, il se déplace de façon aléatoire. Selon le brevet original, lorsqu'il atteint un obstacle, il pivote d'un nombre de degrés aléatoire pour repartir dans une nouvelle direction. Et cependant, il s'acquitte toujours de sa tâche (si vous êtes curieux de voir comment il fonctionne, vous pouvez consulter la page <http://www.explainthatstuff.com/how-roomba-works.html>).

D'un point de vue historique, les algorithmes probabilistes sont une innovation récente, sachant que le premier algorithme de ce type, l'algorithme de recherche des deux points les plus rapprochés (qui détermine, parmi un grand nombre de points d'un plan géométrique, les deux points séparés par la plus petite distance sans devoir les comparer tous) a été mis au point par Michael Rabin en 1976. À ce premier algorithme a succédé l'année suivante le test de primalité aléatoire (un algorithme servant à déterminer si un nombre est composé ou premier), de Robert M. Solovay et Volker Strassen. Peu de temps plus tard, des applications en cryptographie et en informatique distribuée ont rendu la randomisation plus courante et ont fait qu'elle est devenue l'objet d'intenses recherches, et cependant cela reste un domaine nouveau et inconnu.

La randomisation simplifie la recherche d'une solution grâce à un échange entre le temps et la complexité. La simplification des tâches n'est pas son seul avantage : la randomisation permet d'économiser des ressources et elle fonctionne de façon distribuée, avec un besoin réduit de communication et de coordination.

Ce chapitre vous présente l'information nécessaire à la compréhension de la manière dont l'enrichissement de vos algorithmes par un aspect aléatoire vous permet de résoudre des problèmes. D'autres applications encore vous attendent dans les prochains chapitres. C'est pourquoi ce chapitre aborde aussi des sujets essentiels comme les bases des probabilités, les distributions de probabilité et les simulations de Monte Carlo.

Comment fonctionne la randomisation ?

La randomisation repose sur la capacité de l'ordinateur à générer des nombres au hasard, c'est-à-dire à produire l'un après l'autre des nombres sans aucune logique de succession. Un nombre au hasard est donc un nombre imprédictible, et des nombres successifs ne doivent présenter entre eux aucun lien.



Produire un résultat aléatoire n'est cependant pas si facile. Même quand on jette des dés, le résultat ne peut pas être totalement inattendu. Il dépend de la façon dont on tient les dés et dont on les lance, et de leur forme qui n'est pas absolument parfaite. L'ordinateur non plus n'est pas très performant lorsqu'il s'agit de produire des nombres au hasard. Il utilise pour cela des algorithmes ou des séquences numériques pseudo-aléatoires (qui ont pour point de départ une valeur donnée, un nombre équivalent à un indice), car il ne peut pas créer un nombre véritablement aléatoire. L'ordinateur est une machine déterministe : tout ce dont il est constitué procède d'un schéma de réponses bien défini, ce qui signifie qu'il ne peut qu'imiter le hasard d'une manière imparfaite.

Pourquoi a-t-on besoin de la randomisation ?

Même si l'ordinateur ne peut pas produire un résultat véritablement aléatoire, les séquences numériques pseudo-aléatoires (constituées de nombres qui paraissent aléatoires bien qu'étant tout de même prédéterminés d'une certaine manière) peuvent faire l'affaire pour un grand nombre de problèmes d'informatique. Tout algorithme dont la logique fait appel à un processus aléatoire peut être considéré comme un algorithme aléatoire, l'aspect aléatoire pouvant ou non déterminer les

résultats, améliorer la performance ou limiter le risque de ne pas aboutir à une solution dans certains cas.

Généralement, la randomisation sert à la sélection des données d'entrée, elle est le point de départ de l'optimisation, ou bien elle fixe le nombre d'opérations à effectuer sur les données ou leur type. Quand la randomisation devient une partie essentielle de la logique de l'algorithme, et non plus une simple aide à sa performance, le temps d'exécution attendu de l'algorithme et même ses résultats peuvent aussi devenir incertains et aléatoires : par exemple, un algorithme peut produire des résultats différents, bien que de qualité égale, à chaque exécution. Il est donc utile de distinguer les différents types de solutions randomisées, qui portent les noms de lieux célèbres pour leurs jeux de hasard :

- » **Las Vegas** : Ces algorithmes utilisent des inputs ou des ressources aléatoires pour fournir à chaque fois la réponse correcte au problème. Le temps nécessaire à l'obtention d'un résultat est incertain, compte tenu du caractère aléatoire des procédures. L'algorithme Quicksort en est un exemple.
- » **Monte Carlo** : En raison des processus aléatoires sur lesquels ils se basent, les algorithmes de Monte Carlo ne donnent pas toujours une réponse correcte, ni même une réponse, encore qu'une telle situation soit vraiment rare. Le résultat est incertain, mais le temps d'exécution est déterminé. Les algorithmes de Monte Carlo sont la preuve qu'un algorithme ne résout pas toujours le problème qu'il est censé résoudre. Le test de primalité de Solovay-Strassen en est un exemple.
- » **Atlantic City** : Ces algorithmes s'exécutent dans un temps polynomial et donnent une réponse correcte dans au moins 75 % des cas, tandis que les algorithmes de Monte Carlo sont toujours rapides mais ne donnent pas toujours un résultat correct, et que les algorithmes de Las Vegas donnent toujours une réponse correcte mais ne sont pas toujours rapides. Les algorithmes d'Atlantic City sont donc

généralement considérés comme se situant à mi-chemin entre les deux autres types d'algorithmes, sachant que le plus souvent, ils sont rapides et donnent une réponse correcte. Cette classe d'algorithmes a été définie en 1982 par J. Finn dans un manuscrit non publié intitulé *Comparison of Probabilistic Test for Primality*. Créée pour des raisons théoriques en vue de déterminer quels nombres sont premiers, cette catégorie comprend des solutions difficiles à élaborer. Il en existe donc très peu aujourd'hui.

Comprendre comment fonctionnent les probabilités

Une probabilité est l'indication chiffrée des chances qu'un événement se produise. Dans ce livre, et plus généralement dans le domaine des études probabilistes, la probabilité d'un événement est comprise entre 0 (aucune chance que l'événement se produise) et 1 (certitude que l'événement se produira). Une valeur intermédiaire comme 0,25 ou 0,75 indique que l'événement se produira à une certaine fréquence dans les conditions devant y conduire (les *essais*). Même si une plage numérique comprise entre 0 et 1 ne semble pas nécessairement logique à première vue, elle se justifie pour des raisons de simplicité de travail. Une probabilité de 0,25 signifie qu'au cours de 100 essais, l'événement se produira $0,25 * 100 = 25$ fois.

Si la probabilité que votre équipe sportive préférée gagne est de 0,75, par exemple, vous pouvez déterminer ses chances de succès quand elle affronte une autre équipe. Vous pouvez même obtenir des informations plus spécifiques, comme la probabilité de gagner une certaine compétition (par exemple, votre équipe a une probabilité de 0,65 de gagner une partie) ou comme une probabilité conditionnée à un autre événement (par exemple la probabilité que votre équipe gagne n'est plus que de 0,60 quand elle ne joue pas sur son propre terrain).

Les probabilités peuvent vous en apprendre beaucoup sur un événement, et elles sont utilisées aussi par les algorithmes. Dans le cadre d'une approche algorithmique randomisée, vous pouvez vous demander quand un programme doit-il être arrêté, car il devrait avoir déjà abouti à une solution. Il est préférable de savoir combien doit durer l'attente d'une solution avant d'abandonner. Les probabilités peuvent vous permettre de déterminer le nombre d'itérations nécessaires. L'étude du problème 2-SAT au [Chapitre 18](#) est un exemple concret d'utilisation des probabilités comme règles d'arrêt d'un algorithme.



Les probabilités sont souvent exprimées en pourcentages, notamment dans le monde du sport et dans le domaine de l'économie, et elles indiquent le nombre d'occurrences d'un événement au cours de 100 essais. Une probabilité de 0,25 et une probabilité de 25 %, c'est exactement la même chose. C'est simplement une question de convention. Quand on parle de chances ou de risques, c'est une autre manière d'exprimer la probabilité qu'un événement se produise (par exemple, la probabilité qu'un certain cheval gagne la course) en la comparant à la probabilité qu'il ne se produise pas. Dans cet exemple, la probabilité de 0,25 s'oppose à une probabilité de 0,75.

On peut multiplier une probabilité par un nombre d'essais pour estimer le nombre d'occurrences d'un événement, mais le processus inverse permet d'estimer une probabilité de façon empirique : effectuer un certain nombre d'essais, observer les résultats et compter le nombre de fois que l'événement en question se produit. Ce nombre d'occurrences rapporté au nombre d'essais donne l'estimation de la probabilité. En tirant une carte au hasard d'un jeu de 52 cartes, par exemple, la probabilité d'obtenir un trèfle est de 0,25. En effet, un jeu de cartes comporte un nombre égal de piques, de trèfles, de cœurs et de carreaux. La probabilité de tirer un as, sachant qu'il y en a quatre dans le jeu, est de $4 / 52 = 0,077$.



On peut obtenir une estimation plus fiable d'une probabilité empirique en effectuant un plus grand nombre d'essais. Un petit nombre d'essais ne permet pas d'obtenir une estimation fiable de

la probabilité de l'événement, en raison de l'influence de la chance. À mesure que le nombre d'essais augmente, le nombre d'observations, rapporté au nombre d'essais, se rapproche de la probabilité véritable de l'événement. Pour comprendre ce processus générateur, il faut un grand nombre d'essais. Effectuer des essais de cette manière est ce que l'on appelle *l'échantillonnage* d'une distribution de probabilité.

Les distributions de probabilité

La notion de distribution de probabilité est importante également pour améliorer les algorithmes. Une distribution est un tableau de valeurs ou une fonction mathématique qui à toute valeur possible d'un input associe la probabilité que cette valeur apparaisse comme résultat.

Généralement (mais pas toujours), une distribution de probabilité est représentée par un graphique dont l'axe des abscisses représente les valeurs possibles en entrée, et dont l'axe des ordonnées représente la probabilité de l'occurrence. La plupart des modèles statistiques sont basés sur une distribution normale, c'est-à-dire sur une distribution qui est symétrique et dont la forme caractéristique est celle d'une courbe en cloche. Représenter une distribution normale dans Python ([Figure 17-1](#)) ne nécessite que quelques lignes de code (vous trouverez ce code dans le fichier téléchargeable A4D ; 17 ; Probability.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

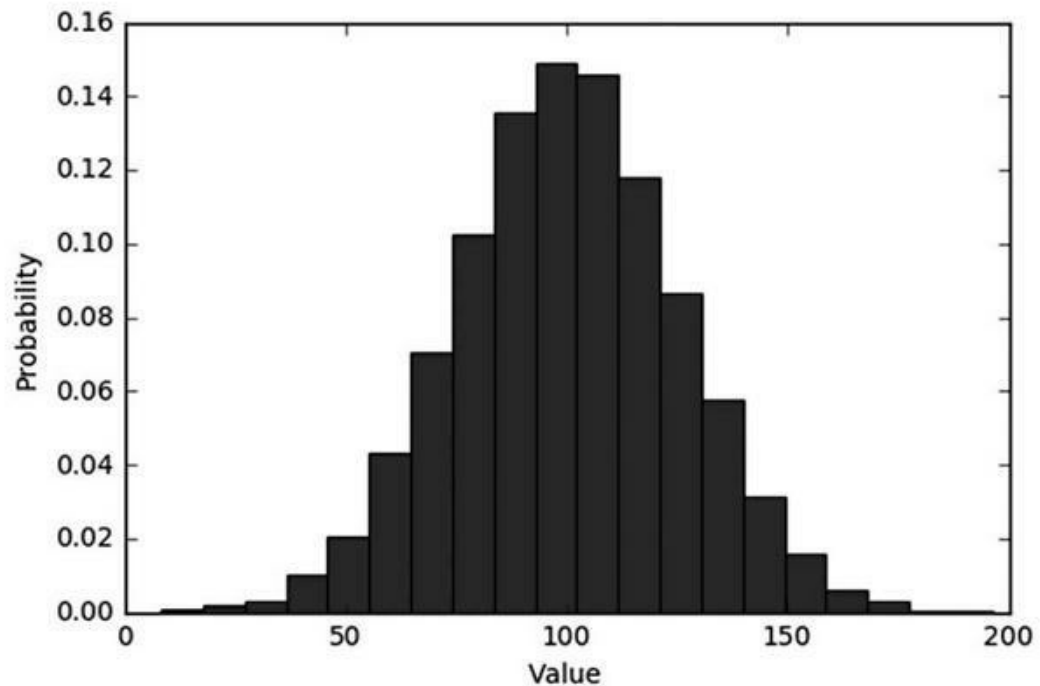


FIGURE 17-1 Histogramme d'une distribution normale.

```
import numpy as np
from numpy.random import normal, uniform
import matplotlib.pyplot as plt
%matplotlib inline

normal_distribution = normal(size=10000) * 25 + 100
weights = np.ones_like(normal_distribution)
           / len(normal_distribution)

plt.hist(normal_distribution, bins=20,
weights=weights)
plt.xlabel("Valeur")
plt.ylabel("Probabilité")
plt.show()
```

La distribution illustrée ici représente un input de 10 000 nombres dont la moyenne est voisine de 100. Chaque barre de l'histogramme représente la probabilité qu'une certaine plage de valeurs apparaisse. La somme de toutes les barres donne la valeur 1, somme de toutes les probabilités exprimées par cette distribution.

Dans une distribution normale, la plupart des valeurs sont plus ou moins proches de la moyenne. Si l'on tire un nombre au hasard parmi les données d'entrée, on a donc de bonnes chances qu'il se situe vers le centre de la distribution. Toutefois, il est possible, même si c'est nettement moins probable, que ce soit un nombre éloigné du centre. Si votre algorithme est plus efficace en utilisant la moyenne qu'en utilisant un autre nombre, le tirage d'un nombre au hasard se justifie et pose sans doute moins de problèmes qu'en poserait une méthode plus subtile pour prélever des valeurs d'entrée.

Une autre importante distribution mentionnée dans ce chapitre est la distribution uniforme. Elle peut aussi être représentée en utilisant un code Python (la [Figure 17-2](#) montre le résultat) :

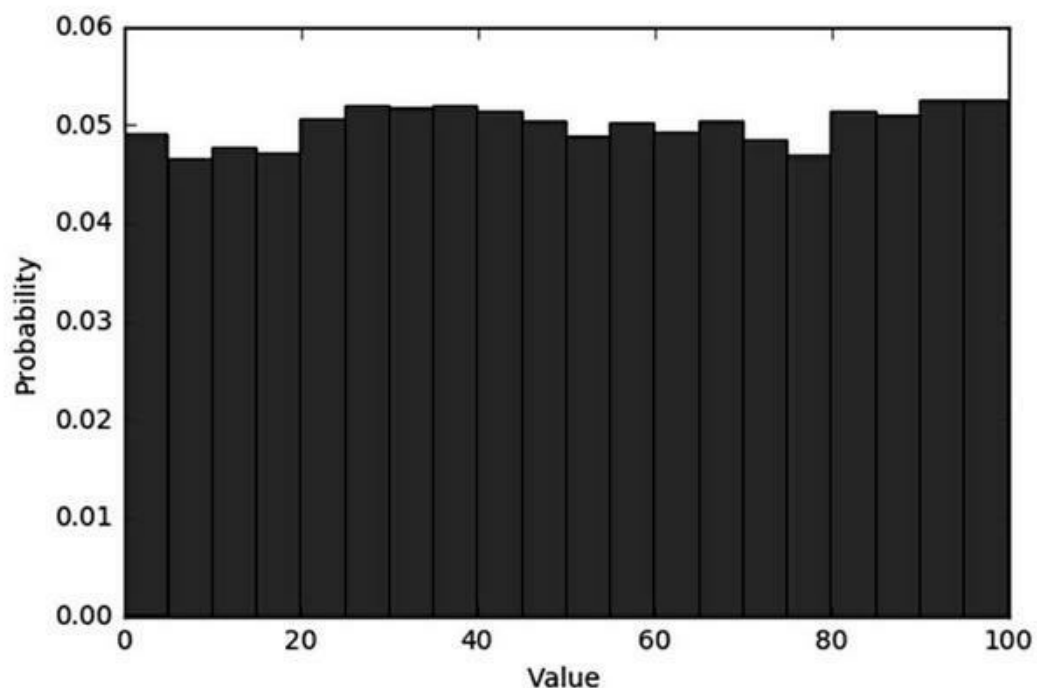


FIGURE 17-2 Histogramme d'une distribution uniforme.

```
uniform_distribution = uniform(size=10000) * 100
weights = np.ones_like(uniform_distribution) / len(uniform_distribution)
plt.hist(uniform_distribution, bins=20, weights=weights)
```

```
plt.xlabel("Valeur")
plt.ylabel("Probabilité")
plt.show()
```

La distribution uniforme est notablement différente de la distribution normale, car chaque nombre a la même probabilité qu'un autre de se trouver dans l'input. En conséquence, les barres de l'histogramme ont à peu près toutes la même taille, et tous les nombres ont la même chance d'être prélevés. C'est un moyen d'éviter de prélever systématiquement le même groupe de nombres quand l'algorithme fonctionne mieux avec des inputs qui varient. Les distributions uniformes sont indiquées, par exemple, quand l'algorithme fonctionne bien avec certains nombres, moyennement avec la plupart, et mal avec quelques autres, et quand on préfère prélever les nombres au hasard pour éviter de se retrouver avec une série de « mauvais » nombres. C'est la méthode qu'utilisent Quickselect et les algorithmes randomisés Quicksort qui sont présentés plus loin dans ce chapitre.

Les algorithmes fonctionnant avec des inputs numériques, connaître leur distribution permet de les faire fonctionner de façon plus intelligente. La distribution initiale n'est pas la seule chose qui compte. Vous pouvez aussi tirer parti de la façon dont la distribution des données change à mesure que le programme se déroule. À titre d'exemple de la manière dont une distribution qui varie permet d'améliorer l'algorithme, le code suivant montre comment deviner une carte tirée au hasard d'un jeu :

```
numbers = ['Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10',
           'Jack', 'Queen', 'King']
seeds = ['Clubs', 'Spades', 'Diamonds', 'Hearts']
deck = [s+'_'+n for n in numbers for s in seeds]

from random import choice
my_cards = deck.copy()
guessed = 0
for card in deck:
    if card == choice(my_cards):
        guessed += 1
print ('Deviné %i carte(s)' % guessed)
```

Deviné 1 carte(s)

Cette méthode donne peu de résultats, et en moyenne on trouve une seule carte sur 52 essais. En effet, pour chaque essai, la probabilité de deviner la carte est égale à $1 / 52$. Une fois qu'on a tiré toutes les cartes, on a donc des chances d'en avoir deviné $(1 / 52) * 52 = 1$. Il est possible de modifier cet algorithme probabiliste simple en éliminant les cartes déjà vues :

```
from random import choice
my_cards = deck.copy()
guessed = 0
for card in deck:
    if card == choice(my_cards):
        guessed += 1
    else:
        my_cards.pop(my_cards.index(card))
print ('Deviné %i carte(s)' % guessed)
```

Deviné 1 carte(s)

À présent, en moyenne, vous devinerez la carte plus souvent, car à mesure que le paquet de cartes se réduit, vos chances de deviner quelle est la prochaine carte tirée augmentent, et vers la fin elles sont élevées (la probabilité est égale à 1 divisé par le nombre de cartes qui restent dans le paquet).



Dans les jeux de cartes, compter les cartes peut donner un avantage. En comptant les cartes et en recourant aux estimations de probabilités, des étudiants du MIT ont gagné des sommes considérables à Las Vegas, jusqu'à ce que cette pratique soit interdite dans les casinos. Cette histoire a même inspiré un film qui est sorti en 2008, dont le titre était *21*, avec Kevin Spacey. Pour plus de détails sur cette histoire, consultez la page <http://www.bbc.com/news/magazine-27519748>.

Simuler l'utilisation de la méthode Monte Carlo

Le calcul des probabilités, en dehors des opérations étudiées précédemment dans ce chapitre, sort du cadre de ce livre. Il n'est pas facile de comprendre le fonctionnement d'un algorithme comportant une part aléatoire, même si l'on sait calculer les probabilités. En effet, ce peut être le résultat du mélange de plusieurs distributions de probabilité différentes. Néanmoins, une étude de la méthode Monte Carlo permet de mieux comprendre les résultats et le fonctionnement des algorithmes les plus complexes. Cette méthode trouve des applications en mathématiques et en physique, pour la résolution d'un certain nombre de problèmes. Des scientifiques comme Enrico Fermi et Edward Teller ont utilisé des simulations de Monte Carlo sur des superordinateurs spécialement conçus pour cela dans le cadre du projet Manhattan (consistant à mettre au point la bombe atomique au cours de la Seconde Guerre mondiale), pour réaliser plus rapidement leurs expériences. Pour plus de détails sur cette application, consultez la page <http://www.atomicheritage.org/history/computing-and-manhattan-project>.



Il ne faut pas confondre la méthode de Monte Carlo avec l'algorithme de Monte Carlo. La première sert à déterminer l'impact d'une distribution de probabilité dans un problème, tandis que le second, comme on l'a vu précédemment, est un algorithme probabiliste avec lequel la détermination d'une solution n'est pas garantie.

La simulation de Monte Carlo consiste à échantillonner les résultats de l'algorithme de façon répétitive. Le programme stocke un certain nombre de résultats puis calcule des statistiques comme la moyenne, et produit une distribution. Ainsi, pour mieux comprendre comment la diminution du paquet de cartes vous permet d'obtenir de meilleurs résultats (comme avec le script Python qui précède), on exécute plusieurs fois l'algorithme et on note le taux de réussite :

```
import numpy as np
samples = list()
for trial in range(1000):
```

```

my_cards = deck.copy()
guessed = 0
for card in deck:
    if card == choice(my_cards):
        guessed += 1
    else:
        my_cards.pop(my_cards.index(card))
samples.append(guessed)

```

L'exécution d'une simulation de Monte Carlo peut durer quelques secondes. Le temps d'exécution dépend de la rapidité de l'algorithme, de la dimension du problème et du nombre d'essais. Cependant, quand on crée des échantillons à partir de distributions, plus on effectue d'essais et plus le résultat est stable. Dans cet exemple, on procède à 1 000 essais. Vous pouvez estimer et visualiser le résultat attendu ([voir Figure 17-3](#)) en utilisant le code suivant :

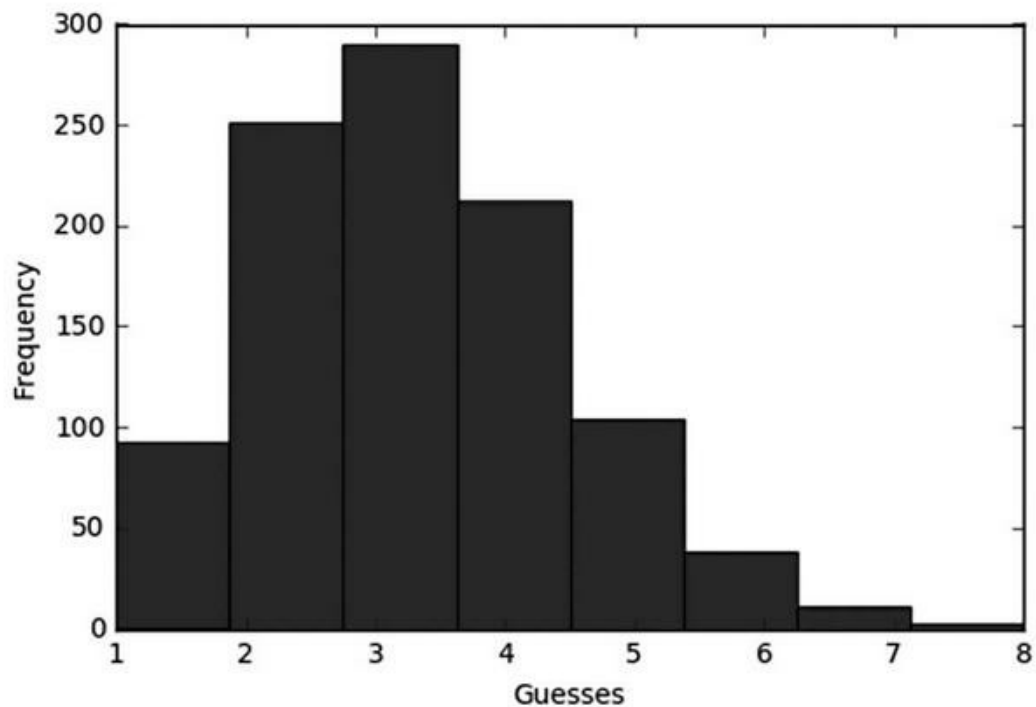


FIGURE 17-3 Affichage des résultats d'une simulation de Monte Carlo.

```

plt.hist(samples, bins=8)
plt.xlabel("Bonnes réponses")
plt.ylabel("Fréquence")

```



```
plt.show()  
print (' En moyenne on peut espérer %0.2f bonnes  
réponses à chaque  
exécution '  
      % np.mean(samples))
```

En moyenne on peut espérer 3.15 bonnes réponses à chaque exécution

En observant l'histogramme résultant, on peut déterminer qu'on obtient un résultat de 3, soit environ 300 tirages sur les 1 000 essais, si bien que la valeur 3 a la plus forte probabilité d'apparaître. Il est intéressant de noter qu'on n'obtient jamais un résultat nul, cependant il est rare de compter sept bonnes réponses ou davantage. Les derniers exemples de ce chapitre utilisent des simulations de Monte Carlo pour étudier le fonctionnement d'algorithmes probabilistes plus élaborés.

Mettre de l'aléa dans votre logique

Voici quelques-unes des nombreuses raisons d'inclure l'aléa dans la logique de votre algorithme :

- » Il améliore le fonctionnement des algorithmes et permet d'obtenir des solutions plus judicieuses.
- » Il diminue le besoin en ressources, en termes de mémoire et de calculs.
- » Il produit des algorithmes qui donnent un output distribué, avec peu ou pas de supervision.

Le prochain chapitre, consacré à la recherche locale, montre l'utilité de la randomisation et des probabilités lorsqu'il est difficile de déterminer l'orientation à donner à l'algorithme. Les exemples des sections qui suivent montrent comment la randomisation permet de trouver rapidement des valeurs sur une certaine position dans les données entrantes et comment la partie aléatoire permet d'accélérer le tri.

Calculer une médiane à l'aide de Quickselect

Le calcul d'un indicateur statistique comme la médiane peut se révéler difficile quand on travaille sur des listes de données non triées. En effet, la médiane dépend de la position des données lorsqu'elles se présentent dans le bon ordre :

- » Si les données d'entrée sont constituées d'un nombre impair d'éléments, la médiane est précisément la valeur située au milieu de leur liste ordonnée.
- » Si les données d'entrée sont constituées d'un nombre pair d'éléments, la médiane est la moyenne des deux nombres dont la paire constitue le milieu de leur liste ordonnée.



À l'instar de la moyenne, la médiane est une valeur unique représentative de la distribution des valeurs observées. Cependant, basée sur l'ordre des éléments du vecteur d'input, la médiane dépend peu des valeurs présentes dans la liste. Elle est simplement la valeur du milieu. En revanche, les valeurs situées aux deux extrémités de la liste des données d'entrée peuvent influencer la moyenne, si ce sont des valeurs extrêmes. Cette robustesse rend la médiane très utile dans diverses situations faisant appel aux statistiques. Un exemple simple de calcul de médiane à l'aide des fonctions de Python vous permettra de mieux le comprendre (vous pouvez retrouver ce code dans le fichier téléchargeable A4D ; 17 ; Quickselect.ipynb sur le site *Dummies* : pour plus de détails, voir l'Introduction).

```
from random import randint, random, choice
import numpy as np
import sys
sys.setrecursionlimit(1500)

n = 501
series = [randint(1,25) for i in range(n)]
print ('La médiane est %0.1f' % np.median(series))
```

La médiane est 14.0

Ce code génère une liste de 501 éléments et trouve la médiane de la liste en utilisant la fonction `median` du module `NumPy`. La médiane trouvée est en réalité la valeur du milieu de la liste ordonnée, c'est-à-dire le 251^e élément :

```
print ('Le 251e élément de la série ordonnée est  
%0.1f' %  
sorted(series)[250])
```

Le 251e élément de la série ordonnée est 14.0

Le tri de la liste et l'extraction de l'élément nécessaire illustrent le fonctionnement de la fonction `median`. Compte tenu de ce tri, le temps d'exécution est de $O(n \cdot \log(n))$ dans le meilleur des cas. Grâce à la randomisation assurée par l'algorithme `Quickselect`, on peut obtenir un résultat meilleur encore, un temps d'exécution de $O(n)$. `Quickselect` fonctionne de façon récursive, c'est pourquoi il faut fixer une limite plus élevée à la récursivité dans Python, compte tenu de la liste et de la position de la valeur cherchée dans la liste ordonnée.

La valeur indice est notée k . L'algorithme procède comme suit :

- 1. Déterminer un nombre pivot dans la liste de données et diviser la liste en deux parties, la liste de gauche étant constituée des nombres inférieurs au nombre pivot, et la liste de droite des nombres supérieurs à ce nombre.**
- 2. Déterminer la longueur de chaque liste. Si la liste de gauche est plus longue que la k -ième position, alors la médiane se trouve dans cette partie. L'algorithme s'applique de façon récursive à cette liste seule.**
- 3. Calculer le nombre de doublons du nombre pivot dans la liste (soustraire de la longueur de la liste la longueur des parties gauche et droite).**
- 4. Déterminer si le nombre de doublons est supérieur à k .**

- a. Si cette condition est vérifiée, cela signifie que l'algorithme a trouvé la solution car la k -ième position fait partie des doublons (c'est le nombre pivot).
- b. Si cette condition n'est pas vérifiée, soustraire de k le nombre de doublons et appliquer le résultat de façon récursive à la partie droite, qui doit contenir la valeur de la k -ième position.

Maintenant que vous avez compris le processus, vous pouvez vous pencher sur quelques lignes de code. L'exemple suivant montre comment utiliser un algorithme de Quickselect.

```
def quickselect(series, k):
    pivot = choice(series)

    left, right = list(), list()
    for item in series:
        if item < pivot:
            left.append(item)
        if item > pivot:
            right.append(item)

    length_left = len(left)
    if length_left > k:
        return quickselect(left, k)
    k -= length_left

    duplicates = len(series) - (length_left +
len(right))
    if duplicates > k:
        return float(pivot)
    k -= duplicates

    return quickselect(right, k)

quickselect(series, 250)
```

14.0

L'algorithme est efficace car il réduit progressivement la dimension du problème. Les meilleurs résultats s'obtiennent quand le nombre pivot se trouve à proximité de la k -ième position

(la règle d'arrêt étant que le nombre pivot est la valeur de la k -ième position). Malheureusement, comme on ne peut pas savoir quelle est la k -ième position dans la liste non ordonnée, procéder de façon aléatoire avec une distribution uniforme (dans laquelle chaque élément de la liste a la même probabilité d'être sélectionné) est la meilleure solution, car l'algorithme finit par trouver la bonne solution. Même lorsque le hasard n'est pas en faveur de l'algorithme, celui-ci continue à réduire le problème, ce qui augmente les chances de trouver la solution, comme on l'a vu précédemment dans ce chapitre lorsqu'il s'agissait de deviner des cartes tirées au hasard d'un paquet. À mesure que le paquet diminuait, il devenait plus facile de deviner la carte. Le code suivant montre comment utiliser Quickselect pour déterminer la médiane d'une liste de nombres :

```
def median(series):
    if len(series) % 2 != 0:
        return quickselect(series, len(series)//2)
    else:
        left = quickselect(series, (len(series)-1) // 2)
        right = quickselect(series, (len(series)+1) //
2)
        return (left + right) / 2

median(series)

14.0
```

Faire des simulations avec Monte Carlo

Dans le cadre d'une familiarisation à l'algorithme Quickselect, il est utile d'en comprendre le fonctionnement interne. Il est possible de contrôler la performance dans différentes conditions en insérant un compteur dans la fonction quickselect et en utilisant une simulation de Monte Carlo :

```
def quickselect(series, k, counter=0):
    pivot = choice(series)
```

```

    left, right = list(), list()
    for item in series:
        if item < pivot:
            left.append(item)
        if item > pivot:
            right.append(item)

    counter += len(series)

    length_left = len(left)
    if length_left > k:
        return quickselect(left, k, counter)
    k -= length_left

    duplicates = series.count(pivot)
    if duplicates > k:
        return float(pivot), counter
    k -= duplicates

    return quickselect(right, k, counter)

```

La première expérience vise à déterminer le nombre d'opérations que l'algorithme doit effectuer, en moyenne, pour trouver la médiane d'une liste de 1001 nombres :

```

results = list()
for run in range(1000):
    n = 1001
    series = [randint(1,25) for i in range(n)]
    median, count = quickselect(series, n//2)
    assert(median==np.median(series))
    results.append(count)

print ("Nombre moyen d'opérations : %i" %
np.mean(results))

```

Nombre moyen d'opérations : 2764

La représentation des résultats sous la forme d'un histogramme ([voir Figure 17-4](#)) fait apparaître que le nombre de calculs effectués par l'algorithme représente entre deux et quatre fois la taille de l'input, le nombre de calculs le plus probable étant autour de trois fois cette taille.

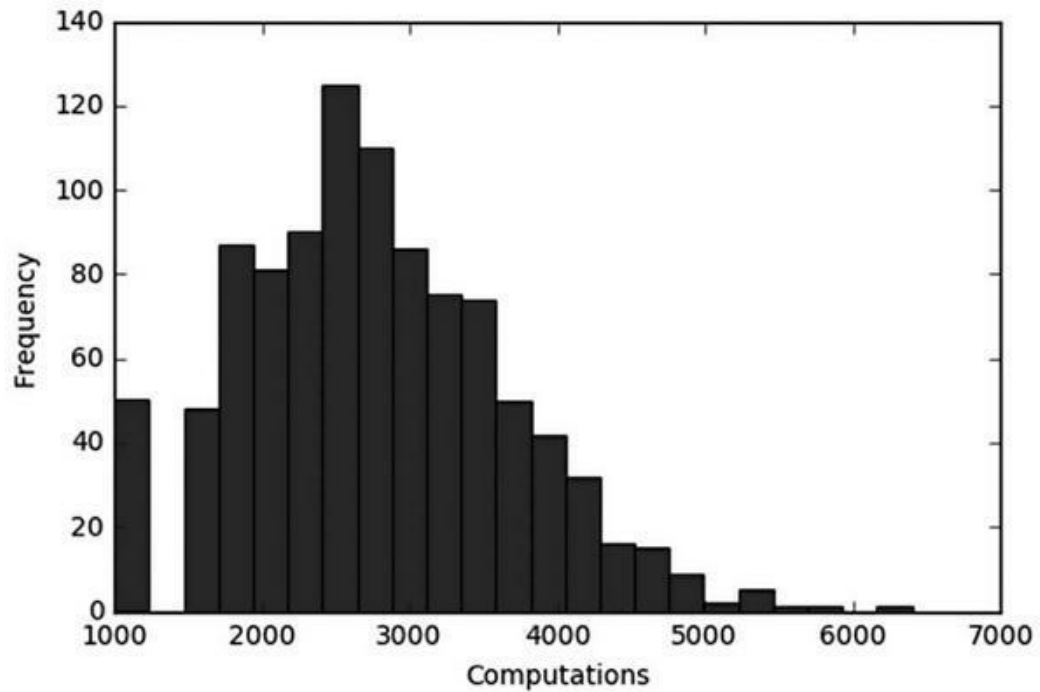


FIGURE 17-4 Résultats d’une simulation de Monte Carlo avec Quickselect.

```
import matplotlib.pyplot as plt

%matplotlib inline

plt.hist(results, bins='auto')
plt.xlabel("Nombre de calculs")
plt.ylabel("Fréquence")
plt.show()
```

S’il faut en moyenne trois fois la taille de l’input, Quickselect se révèle performant. Néanmoins, vous vous demandez peut-être si cette proportion entre le nombre d’inputs et le nombre de calculs continue de se vérifier quand la taille de l’input augmente. Comme on l’a vu à propos des problèmes NP-complets, la croissance de la taille de l’input entraîne une aggravation des problèmes. On peut vérifier cette théorie par une autre simulation de Monte Carlo en complément de la précédente, le résultat étant montré sur la [Figure 17-5](#).

```

input_size = [501, 1001, 5001, 10001, 20001, 50001]
computations = list()
for n in input_size:
    results = list()
    for run in range(1000):
        series = [randint(1, 25) for i in range(n)]
        median, count = quickselect(series, n//2)
        assert(median==np.median(series))
        results.append(count)
    computations.append(np.mean(results))

plt.plot(input_size, computations, '-o')
plt.xlabel("Taille de l'input")
plt.ylabel("Nombre de calculs")
plt.show()

```

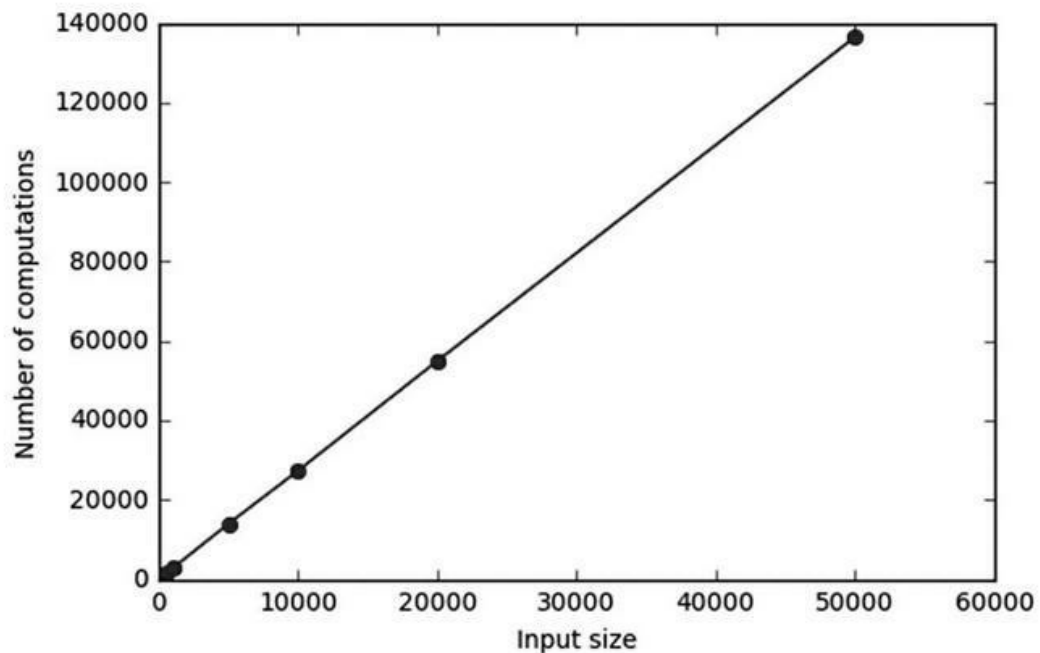


FIGURE 17-5 Représentation des simulations de Monte Carlo quand l'input croît.



Dans cet exemple, les calculs peuvent durer jusqu'à dix minutes (le temps d'exécution de certaines simulations de Monte Carlo peut être vraiment long), mais le résultat permet de bien voir ce que représente l'utilisation d'un algorithme qui fonctionne en temps linéaire. Quand l'input croît (sur l'axe des abscisses), le

nombre de calculs (représenté sur l'axe des ordonnées) augmente de façon proportionnelle, si bien que le graphe correspondant est une droite.

Ordonner plus vite avec Quicksort

Le [Chapitre 7](#) présente les algorithmes de tri, qui sont à la base même de toute la connaissance actuelle en matière d'algorithmes de programmation. L'algorithme Quicksort, qui peut fonctionner en temps logarithmique mais qui ne donne pas toujours un résultat et qui produit des résultats en temps quadratique avec des entrées de données mal conditionnées, vous surprendra sûrement. Cette section étudie les raisons pour lesquelles cet algorithme peut ne pas marcher, et propose une solution efficace consistant à y introduire de l'aléa. Étudions tout d'abord le code suivant :

```
def quicksort(series, get):  
    try:  
        global operations  
        operations += len(series)  
    except:pass  
  
    if len(series) <= 3:  
        return sorted(series)  
  
    pivot = get(series)  
    duplicates = series.count(pivot)  
  
    left, right = list(),list()  
    for item in series:  
        if item < pivot:  
            left.append(item)  
        if item > pivot:  
            right.append(item)  
  
    return quicksort(left, get) + [pivot  
    ] * duplicates + quicksort(right, get)
```

Il s'agit d'une autre implémentation de l'algorithme du [Chapitre 7](#). Cette fois, cependant, le code extrait la fonction qui

détermine quel pivot l'algorithme va utiliser pour diviser la liste initiale de façon récursive. L'algorithme détermine la division en partant de la première valeur de la liste. Il détermine aussi le nombre d'opérations nécessaire pour ordonner les données à l'aide de la variable globale `operations`, qui est définie, réinitialisée et accédée en tant que compteur extérieur à la fonction. Le code suivant teste l'algorithme dans des conditions inhabituelles, lorsqu'il doit traiter une liste déjà ordonnée. Remarquez combien il est performant :

```
series = list(range(25))
operations = 0
sorted_list = quicksort(series, choose_leftmost)
print ("Nombre d'opérations : %i" % operations)
```

Nombre d'opérations : 322

Ici, l'algorithme doit exécuter 322 opérations pour ordonner une liste de 25 éléments, ce qui constitue une très mauvaise performance. L'utilisation d'une liste déjà ordonnée est la cause du problème. En effet, l'algorithme divise la liste en deux, ce qui donne une liste vide et une liste contenant les valeurs résiduelles, et il doit répéter cette division inutile pour toutes les valeurs uniques présentes dans la liste. En général, l'algorithme Quicksort fonctionne bien, car il traite des listes non ordonnées, et sélectionner l'élément le plus à gauche équivaut à tirer un nombre au hasard pour en faire le pivot. Pour éviter ce problème, vous pouvez utiliser une variante de l'algorithme donnant un tirage véritablement aléatoire de la valeur pivot.

```
def choose_random(l): return choice(l)

series = [randint(1,25) for i in range(25)]
operations = 0
sorted_list = quicksort(series, choose_random)
print ("Nombre d'opérations : %i" % operations)
```

Nombre d'opérations : 81

À présent l'algorithme exécute la tâche au moyen d'un plus petit nombre d'opérations correspondant précisément au temps d'exécution $n \cdot \log(n)$, c'est-à-dire $25 \cdot \log(25) = 80,5$.

Chapitre 18

Effectuer une recherche locale

DANS CE CHAPITRE

- » **Savoir effectuer une recherche locale sur un problème NP-difficile**
 - » **Faire appel à l'heuristique et aux solutions de voisinage**
 - » **Résoudre le problème 2-SAT par une recherche locale et une randomisation**
 - » **Découvrir les différentes astuces applicables à une recherche locale**
-

Face à un problème NP-difficile, c'est-à-dire à un problème pour lequel aucune solution connue n'est associée à une complexité d'exécution moins qu'exponentielle (voir la discussion sur la théorie des problèmes NP-complets au [Chapitre 15](#)), on dispose de peu d'alternatives envisageables. À partir de l'idée que les problèmes de classe NP impliquent des compromis (comme accepter des résultats partiels ou non optimaux), les options suivantes offrent une solution à ce problème qui, autrement, serait insoluble :

- » Identifier les cas particuliers dans lesquels on peut résoudre le problème efficacement en temps polynomial à l'aide d'une méthode précise ou d'un algorithme glouton. Cette approche simplifie le problème et limite le nombre de combinaisons de solutions à essayer.
- » Employer des techniques de programmation dynamique (voir [Chapitre 6](#)) qui améliorent la recherche par force brute et réduisent la complexité du problème.

- » Rechercher un compromis et élaborer un algorithme d'approximation capable de trouver une solution partielle, proche de l'optimum. Lorsque vous êtes satisfait de la solution partielle, vous réduisez le temps d'exécution de l'algorithme. Les algorithmes d'approximation peuvent être :
 - des algorithmes gloutons (voir [Chapitre 15](#)) ;
 - des algorithmes de recherche locale utilisant la randomisation ou une autre technique heuristique (c'est le sujet du présent chapitre) ;
 - des algorithmes de programmation linéaire (c'est le sujet du [Chapitre 19](#)).
- » Choisir une heuristique ou une *métaheuristique* (une heuristique qui vous permet de déterminer quelle heuristique vous devez utiliser) qui soit bien adaptée à votre problème dans la pratique. Cependant, cette approche n'offrira aucune garantie théorique et elle sera plutôt empirique.

La recherche locale, qu'est-ce que c'est ?

La *recherche locale* est une méthode générale de résolution des problèmes faisant appel à un vaste ensemble d'algorithmes, permettant d'éviter les complexités exponentielles de nombreux problèmes de type NP. Une recherche locale prend pour point de départ une solution imparfaite et s'en éloigne par étapes successives. Elle détermine la viabilité de solutions voisines et peut aboutir à la solution parfaite, en se fondant sur un choix aléatoire ou sur une heuristique astucieuse (aucune méthode exacte n'entre en jeu).



Une *heuristique* est une estimation éclairée concernant une solution, comme par exemple une règle simple pour se diriger vers un résultat désiré, mais sans pouvoir dire exactement comment y parvenir. C'est comme être perdu dans une ville que l'on ne connaît pas et se faire indiquer un certain chemin pour retrouver l'hôtel (mais sans instructions précises), ou simplement la distance à laquelle on se trouve de l'hôtel. Certaines solutions de recherche locale font appel à l'heuristique, et vous les trouverez donc dans ce chapitre. Le [Chapitre 20](#) étudie en détail l'utilisation d'une heuristique pour exécuter des tâches pratiques.

Vous n'avez aucune garantie qu'une recherche locale aboutisse à une solution au problème, mais vos chances augmentent à partir du moment où vous consacrez assez de temps à la recherche pour qu'elle exécute les calculs nécessaires. Le programme ne s'arrête que lorsqu'il ne trouve plus aucun moyen d'améliorer la solution à laquelle il a abouti.

Faire le tour du voisinage

Les algorithmes de recherche locale procèdent par itérations pour améliorer une solution de départ, en progressant étape par étape et en déterminant des solutions voisines jusqu'à ce qu'il ne leur soit plus possible d'améliorer la solution. Les algorithmes de recherche locale étant aussi simples et intuitifs que les algorithmes gloutons, concevoir une méthode de recherche locale pour résoudre un problème d'algorithmique n'est pas difficile. Il suffit de définir la bonne procédure :

1. **Commencer par une solution existante (généralement une solution aléatoire ou une solution donnée par un autre algorithme).**
2. **Rechercher une série de nouvelles solutions possibles au voisinage de la solution courante, constituant ainsi une liste de solutions candidates.**

3. **Déterminer laquelle de ces solutions doit être utilisée à la place de la solution courante, d'après le résultat d'une heuristique acceptant en entrée la liste des solutions candidates.**
4. **Poursuivre l'exécution des étapes 2 et 3 jusqu'à ce qu'il n'apparaisse plus aucune amélioration possible de la solution courante, ce qui signifie que l'on a abouti à la meilleure solution disponible.**



La recherche locale de solutions, même si elle est facile à concevoir, ne permet pas nécessairement de trouver une solution dans un temps raisonnable (on peut arrêter le processus et se servir de la solution courante) ou de produire une solution de qualité minimum. Vous pouvez utiliser quelques trucs du métier pour tirer le meilleur profit de cette approche.

Au début de la recherche locale, vous choisissez une solution initiale. Si vous optez pour une solution au hasard, il est utile d'englober la recherche dans des itérations répétées pour produire différentes solutions de départ aléatoires. Parfois, la possibilité de parvenir en fin de compte à une bonne solution dépend du point de départ. Si vous partez d'une solution existante en vue de l'affiner, y appliquer un algorithme glouton peut se révéler un bon compromis pour obtenir une solution pas trop longue à produire.

Après avoir choisi un point de départ, définissez le voisinage et déterminez sa dimension. Pour définir un voisinage, il faut se représenter le plus petit changement que l'on peut appliquer à la solution. Lorsqu'une solution est un ensemble d'éléments, toutes les solutions voisines sont les ensembles dont un des éléments change. Dans le problème du voyageur de commerce, par exemple, les solutions voisines peuvent consister à changer les destinations finales de deux (ou plusieurs) trajets ([Figure 18-1](#)).

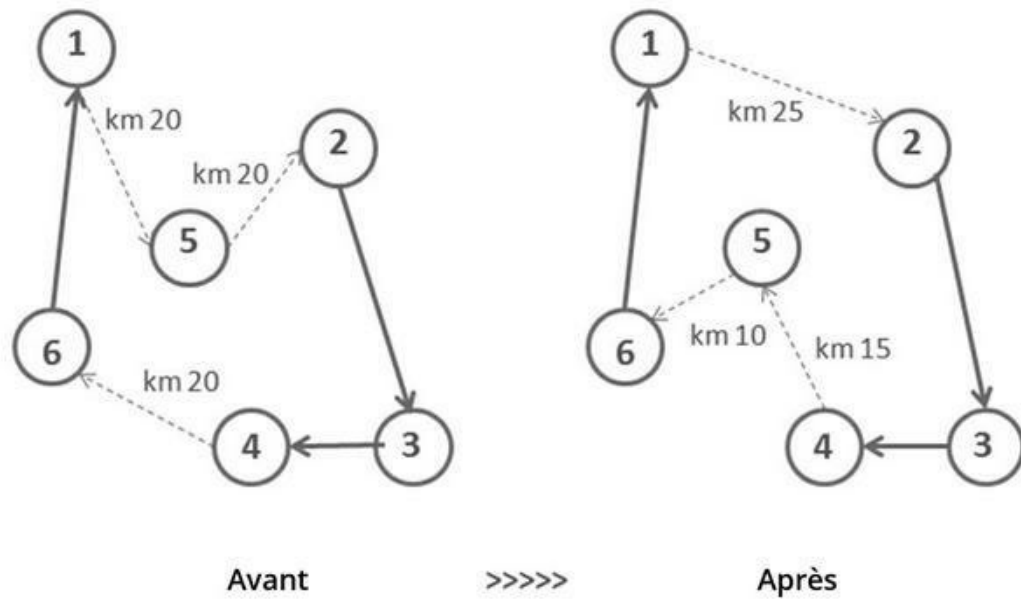


FIGURE 18-1 Dans un problème de voyageur de commerce, échanger les trajets finals peut permettre d’obtenir de meilleurs résultats.

Selon la manière dont vous allez définir le voisinage, vous aurez des listes candidates plus courtes ou plus longues. Des listes plus longues impliquent davantage de temps et de calculs, mais contrairement aux listes courtes, elles offrent davantage de possibilités que le processus se termine plus tôt et avec un meilleur résultat.

La longueur des listes représente un compromis que vous pouvez améliorer en recourant à une expérimentation après chaque test, pour déterminer si allonger ou raccourcir la liste candidate apporte un avantage ou un désavantage en termes de temps de réponse et de qualité de la solution.

Basez le choix de la nouvelle solution sur une heuristique et, compte tenu du problème, déterminez la meilleure solution. Dans le problème du voyageur de commerce, par exemple, échangez les trajets de manière à réduire le plus possible la longueur du trajet total. Dans certains cas, vous pouvez utiliser une solution aléatoire plutôt qu’une heuristique (comme on va le voir à propos du problème SAT-2 dans ce chapitre). Même avec une heuristique limpide, l’algorithme peut trouver plusieurs bonnes solutions.

L'ajout d'un aléa peut rendre votre recherche locale plus efficace. Face à plusieurs solutions, vous pouvez sans risque en choisir une au hasard.



En principe, dans une recherche locale, on obtient les meilleurs résultats en lançant un certain nombre de recherches, et en donnant à la solution de départ ainsi qu'aux étapes suivantes le caractère le plus aléatoire possible. Ne comptez sur l'heuristique que lorsqu'elle présente un avantage évident. La recherche locale et l'aléatoire vont bien ensemble.

Votre recherche locale doit s'arrêter à un certain point, aussi devez-vous choisir les règles d'arrêt : lorsque votre heuristique ne peut plus trouver de bonnes solutions voisines, ou lorsqu'elle ne peut plus améliorer la qualité de la solution (par exemple en calculant une fonction de coût, comme dans l'exemple du voyageur de commerce avec le calcul de la longueur total de la tournée). Si vous ne fixez pas une règle d'arrêt adaptée au problème à résoudre, votre recherche risque de ne jamais s'arrêter ou de durer un temps inacceptable. Au cas où vous ne pourriez pas en fixer une, fixez simplement une limite de temps ou limitez le nombre d'essais. Comptez les essais et décidez, à un moment donné, qu'il n'est pas utile de poursuivre, car la probabilité de succès devient trop faible.

Les trucs de la recherche locale

La recherche locale consiste à partir d'une solution courante pour progresser vers des solutions voisines, une par une, jusqu'à trouver une bonne solution (ou jusqu'à s'arrêter à la solution qu'il n'est plus possible d'améliorer). Cette méthode présente d'importants avantages quand on cherche à résoudre des problèmes NP-difficiles, pour les raisons suivantes :

- » Elle est simple à concevoir et à appliquer.
- » Elle utilise peu de mémoire et de ressources informatiques (mais les recherches demandent du temps d'exécution).

- » Elle aboutit à des solutions acceptables, voire bonnes en partant d'une solution imparfaite (les solutions voisines constituant un chemin vers la solution ultime).



Un problème qu'une recherche locale permet de résoudre peut être assimilé à un graphe de solutions interconnectées. L'algorithme parcourt le graphe en progressant de sommet en sommet et en recherchant le sommet qui satisfera aux exigences de la tâche. De ce point de vue, la recherche locale tire parti d'algorithmes d'exploration de graphe comme le parcours en profondeur (DFS) ou le parcours en largeur (BFS), l'un et l'autre étudiés au [Chapitre 9](#).

La recherche locale est un moyen viable de trouver des solutions acceptables à des problèmes NP-difficiles. Cependant, elle ne peut pas fonctionner correctement sans la bonne heuristique. La randomisation peut constituer un bon complément à la recherche locale, avec les possibilités suivantes :

- » **Échantillonnage aléatoire** : Génération de solutions de départ.
- » **Marche aléatoire** : Sélection d'une solution aléatoire voisine de la solution courante (pour plus de détails sur les marches aléatoires, lire la section « Résoudre 2-SAT grâce à la randomisation », plus loin dans ce chapitre).

La randomisation n'est pas la seule heuristique disponible. Une recherche locale peut se fonder sur une exploration de solutions plus rationnelle en utilisant une fonction objectif pour s'orienter (comme dans l'optimisation par la méthode de l'*escalade*) et éviter le piège des solutions couci-couça (comme le *recuit simulé* et la *recherche tabou*). Une *fonction objectif* est une computation permettant d'évaluer la qualité de la solution en produisant un score. S'il vous faut des scores plus élevés avec l'*escalade*, ou *hill climbing*, c'est que vous êtes confronté à un problème de maximisation ; si vous recherchez des scores moins élevés, il s'agit d'un problème de minimisation.

Expliquer l'escalade avec les dames des échecs

Il est facile de trouver des analogies concernant les techniques employées par la recherche locale, sachant que de nombreux phénomènes mettent en jeu une transition graduelle d'une situation vers une autre. La recherche locale n'est pas simplement une technique conçue par des spécialistes des algorithmes, c'est en réalité un processus observable dans la nature et dans la société humaine. Dans la société et dans la science, par exemple, l'innovation peut être considérée comme une recherche locale de la prochaine étape parmi les technologies actuellement accessibles :

<https://www.technologyreview.com/s/603366/mathematical-model-reveals-the-patterns-of-how-innovations-arise/>.

Les heuristiques sont souvent issues du monde physique : elles sont inspirées de la force de gravitation, de la fusion des métaux, de l'évolution de l'ADN chez les animaux ou du comportement des fourmis, des abeilles et des lucioles (l'article de la page <https://arxiv.org/pdf/1003.1464.pdf> explique l'algorithme de Lévy).

L'escalade est inspirée de la force de la gravitation. Elle repose sur la constatation que lorsqu'un ballon dévale une pente, il suit le chemin le plus pentu, et quand il est lancé dans une côte, il a tendance à suivre la trajectoire la plus directe pour atteindre le sommet. Graduellement, une étape après l'autre, qu'il monte ou qu'il descende, le ballon arrive à destination, là où il n'est plus possible de monter ou de descendre.

Dans une recherche locale, on peut reproduire avec succès la même procédure en utilisant une *fonction objectif* qui évalue les solutions voisines et détermine laquelle constitue une amélioration de la solution courante. Pour poursuivre l'analogie avec l'escalade, utiliser une fonction objectif est comme sentir l'inclinaison du terrain et déterminer le meilleur mouvement pour continuer le parcours. À partir de sa position, le randonneur

évalue chaque direction en vue de déterminer l'inclinaison du terrain. Si son objectif est de parvenir au sommet, il choisira la direction correspondant à la plus forte pente ascendante. Cependant, il ne s'agit là que de la situation idéale : souvent, les randonneurs rencontrent des difficultés au cours de leur ascension et doivent trouver d'autres solutions pour les contourner.

Une fonction objectif est similaire à un critère d'avidité (voir [Chapitre 15](#)). Elle ne prend en compte que la destination, si bien qu'elle peut déterminer la direction à prendre, mais ne détecte pas les obstacles. Songeons à l'effet de cécité lors de l'ascension d'une montagne : il est difficile de dire à quel moment un randonneur atteindra le sommet. Un terrain plat, sans possibilité de continuer à monter, peut indiquer que le sommet est atteint. Cependant, un endroit plat peut aussi être une plaine, une portion de chemin à plat, ou même un trou dans lequel le randonneur pourrait être tombé. Dans la mesure où le randonneur est aveugle, on ne peut avoir aucune certitude.

Le même problème se pose quand une recherche locale s'appuie sur une heuristique d'escalade : le programme recherche progressivement de meilleures solutions jusqu'à ce qu'il ne puisse plus en trouver quand il examine les solutions qui existent au voisinage de la solution courante. L'algorithme déclare alors avoir trouvé la solution. Il la présente aussi comme la solution globale, même si, comme l'illustre la [Figure 18-2](#), il a simplement trouvé une solution locale maximale, qui n'est la meilleure dans ce voisinage que parce que celles qui l'entourent sont pires. Il reste possible de trouver une meilleure solution en poursuivant l'exploration.

Un exemple d'utilisation de la méthode de l'escalade (avec le risque de se retrouver bloqué dans un maximum local, ou dans un minimum local quand on descend, comme dans cet exemple) est le problème des n dames (n -queens), inventé par le spécialiste des échecs Max Bezzel en 1848 à l'attention des amateurs de jeux d'échecs. Il s'agit de placer n dames sur un échiquier de dimensions $n \times n$, de telle sorte qu'aucune ne soit menacée par une

autre (sachant qu'aux échecs, une dame peut se déplacer et prendre une pièce aussi bien en ligne qu'en colonne ou en diagonale).

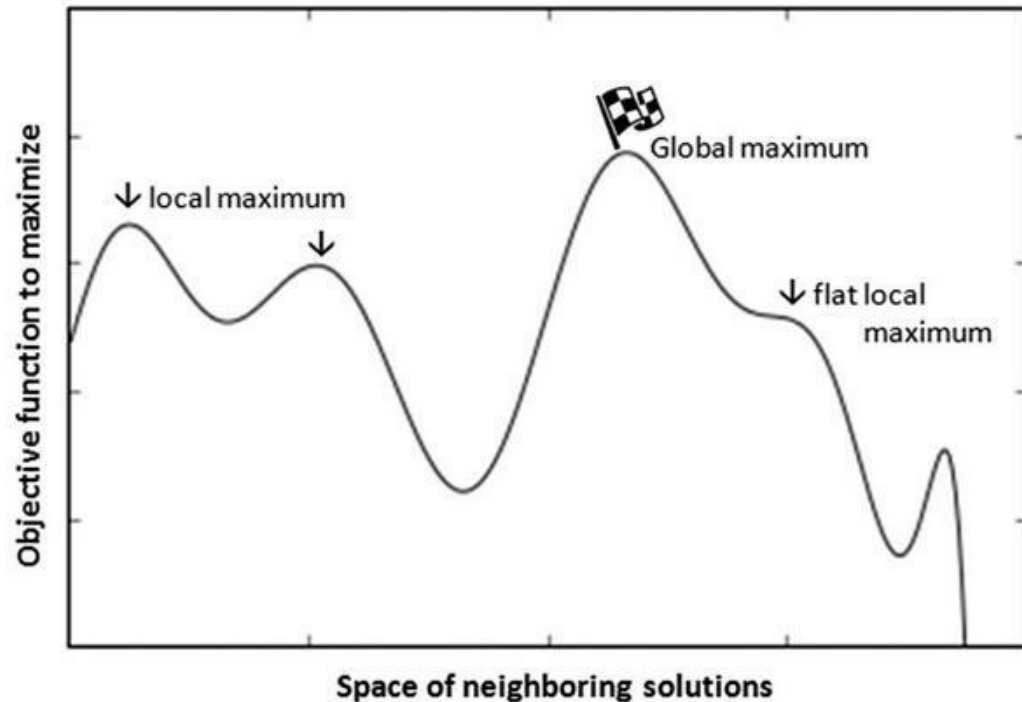


FIGURE 18-2 La recherche locale explore les environs par la méthode de l'escalade.



Il s'agit bien d'un problème NP-difficile. Il existe 4 426 165 368 façons différentes de placer huit dames sur un échiquier de 8 x 8 cases, parmi lesquelles 92 configurations seulement sont des solutions à ce problème. À l'évidence, ce problème ne peut pas être résolu par la force brute, ni par la seule chance. En revanche, la recherche locale permet de le résoudre très simplement, en recourant à la méthode de l'escalade :

1. **Placer les n dames sur l'échiquier au hasard, mais chacune sur une colonne différente (qu'il n'y en ait pas deux sur la même colonne).**
2. **Évaluer le prochain ensemble de solutions en déplaçant chaque reine d'une case sur cette colonne, dans un sens**

ou dans l'autre. Cette étape suppose $2 * n$ déplacements.

3. Déterminer le nombre de dames qui en menacent une autre après chaque déplacement.
4. Déterminer avec quelle solution le nombre de dames qui en menacent une autre est le plus réduit, et retenir cette solution pour la prochaine itération.
5. Exécuter les étapes 2 à 4 jusqu'à aboutir à une solution.

Malheureusement, cette méthode n'est efficace que dans 14 % des cas environ, car dans 86 % des cas, on se retrouve bloqué dans une configuration qui ne permet plus aucune amélioration (le nombre de dames menacées ne diminue avec aucun des $2 * n$ déplacements envisageables en tant que solutions voisines). Le seul moyen d'échapper à cette situation est de reprendre à zéro la recherche locale en repartant d'une autre configuration aléatoire. La [Figure 18-3](#) représente une solution réussie.

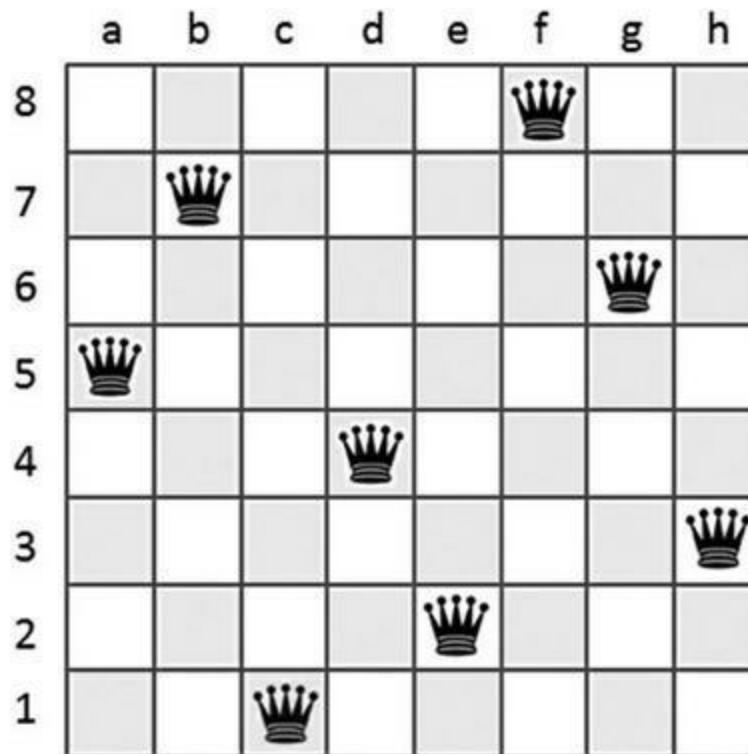


FIGURE 18-3 Une solution au problème des 8 dames.

Malgré cette faiblesse, les algorithmes d'escalade sont utilisés partout, et plus particulièrement dans les domaines de l'intelligence artificielle et de l'apprentissage machine. Les réseaux neuronaux qui reconnaissent des sons ou des images, les téléphones mobiles et les véhicules sans chauffeur dépendent principalement d'un système d'optimisation par la méthode de l'escalade appelé le *gradient conjugué*. Les démarrages randomisés et l'ajout d'aléa dans la procédure d'escalade permettent de ne pas rester piégé dans une solution locale et d'atteindre le maximum global. Le recuit simulé et la recherche tabou sont deux manières judicieuses d'exploiter les décisions aléatoires au cours de l'escalade.

Le recuit simulé, qu'est-ce que c'est ?

À un certain moment de la recherche, si votre fonction objectif ne vous donne plus les bonnes indications, vous pouvez recourir à une autre heuristique pour contrôler la situation et tenter de trouver un meilleur chemin vers une meilleure solution. C'est ainsi que fonctionnent le recuit simulé et la recherche tabou : chacune de ces deux méthodes est une issue de secours.

Le recuit simulé tire son nom d'une technique utilisée en métallurgie, le recuit, consistant à chauffer le métal à haute température, puis à le laisser refroidir lentement afin de l'assouplir pour le travailler à froid et éliminer les défauts internes de cristallisation (pour plus de détails, voir <http://www.brighthubengineering.com/manufacturing-technology/30476-what-is-heat-treatment/>). La recherche locale reprend ce principe en assimilant la recherche de solution à une structure atomique qui évoluerait de manière à devenir plus maniable. La température est l'élément qui change la donne dans le processus d'optimisation. De même que les hautes températures assouplissent la structure du matériau (les solides fondent et les

liquides s'évaporent), les hautes températures dans un algorithme de recherche locale entraînent un assouplissement de la fonction objectif, si bien que les moins bonnes solutions peuvent être préférées aux meilleures. Le recuit simulé modifie la procédure d'escalade en réservant la fonction objectif pour l'évaluation des solutions voisines, mais en l'utilisant pour déterminer le choix d'une façon différente :

- 1. Obtenir une température exprimée sous la forme d'une probabilité (en physique, la fonction de Gibbs-Boltzmann est une formule qui convertit la température en probabilité. Expliquer cette fonction sort du cadre de ce livre, mais vous pouvez la découvrir sur la page <http://www.iue.tuwien.ac.at/phd/binder/node87.html>).**
- 2. Fixer un programme de température. La température décroît à une certaine vitesse au cours du temps et au cours de l'exécution de la recherche.**
- 3. Définir une solution de départ (à l'aide d'un échantillonnage aléatoire ou d'une autre technique) et démarrer une boucle. À mesure que la boucle s'exécute, la température décroît.**
- 4. Arrêter l'optimisation quand la température atteint la valeur zéro.**
- 5. Proposer le résultat courant comme solution.**

À ce stade, il faut réitérer la recherche de solutions. À chaque étape de l'itération précédente, entre les étapes 3 et 4 qui précèdent, il convient de procéder comme suit :

- 1. Lister les solutions voisines et en choisir une au hasard.**
- 2. Si la solution voisine est meilleure que la solution courante, cette solution voisine devient la solution courante.**
- 3. Dans le cas contraire, choisir un nombre au hasard entre 0 et 1 sur la base d'un seuil de probabilité associé à**

la température, et déterminer s'il est inférieur à ce seuil :

- S'il est inférieur, prendre la solution voisine comme solution courante (même si elle est moins bonne que la solution courante d'après la fonction objectif).
- S'il est supérieur, conserver la solution courante.

Le recuit simulé est un moyen judicieux d'améliorer l'escalade, car il évite de devoir cesser la recherche pour se contenter d'une solution locale. Quand la température est suffisamment élevée, la recherche peut utiliser une solution aléatoire et trouver une autre voie pour parvenir à une meilleure optimisation. La température étant élevée au début de la recherche, l'algorithme a la possibilité d'ajouter de l'aléa à l'optimisation. À mesure que la température descend et se rapproche de zéro, l'algorithme a de moins en moins de chances de pouvoir sélectionner une solution aléatoire, et la recherche locale procède comme dans l'escalade. Dans le problème du voyageur de commerce, par exemple, l'algorithme effectue un recuit simulé en remettant en cause la solution courante à haute température :

- » en effectuant la sélection aléatoire d'un segment de la tournée et en le parcourant en sens inverse ;
- » et en visitant une ville plus tôt ou plus tard au cours de la tournée, l'ordre de visite des autres villes restant inchangé.

Quand les ajustements qui en résultent aboutissent à allonger le trajet total de la tournée, l'algorithme les conserve ou les rejette selon la température au cours du processus de recuit simulé.

Éviter de répéter l'utilisation de la recherche tabou

Tabou est un mot ancien du tongan, une langue polynésienne, désignant certaines choses auxquelles on ne doit pas toucher parce qu'elles sont sacrées. Le mot *tabou* est sorti du domaine des

études anthropologiques pour gagner le langage courant et désigner ce qui est interdit. Dans le domaine de l'optimisation par la recherche locale, on se retrouve souvent coincé dans un voisinage de solutions qui n'offrent aucun progrès, c'est-à-dire avec une solution locale qui semble être la meilleure solution, mais qui est loin d'être la solution voulue. La recherche tabou consiste à assouplir certaines règles, tout en en appliquant d'autres afin de pouvoir sortir d'un optimum local et d'aboutir à de meilleures solutions.

L'heuristique de la recherche tabou englobe les fonctions objectifs et progresse en passant en revue un certain nombre de solutions voisines. Elle intervient quand vous ne pouvez plus progresser car les solutions voisines ne vous rapprochent plus de votre objectif. Quand cela se produit, la recherche tabou procède comme suit :

- » Elle permet le recours à une solution insatisfaisante le temps de voir s'il est possible, en s'éloignant de la solution locale, de trouver un meilleur chemin conduisant à la meilleure solution.
- » Elle garde en mémoire les solutions essayées et les interdit, afin d'éviter que la recherche boucle entre les mêmes solutions voisines de la solution locale sans trouver une échappée.
- » Elle crée une mémoire à long ou à court terme des solutions tabou en changeant la longueur de la file utilisée pour stocker les solutions qui ont précédé. Quand la file est remplie, l'heuristique abandonne le résultat de la recherche tabou la plus ancienne afin de faire place à la nouvelle.

La recherche tabou s'apparente à la mise en cache et à la mémorisation (voir [Chapitre 16](#)). Elle nécessite que l'algorithme conserve la trace des étapes parcourues afin de gagner du temps et d'éviter de repasser par des solutions déjà utilisées. Dans le problème du voyageur de commerce, elle peut être utile à l'optimisation de la solution en changeant l'ordre de visite de

deux ou plusieurs villes et en évitant la répétition de certaines séries de solutions.

Résoudre la satisfiabilité des circuits booléens

À titre d'illustration pratique de la méthode de la recherche locale, cet exemple classique de problème NP-complet étudie la satisfiabilité d'un circuit au moyen d'un algorithme de randomisation et de simulation de Monte Carlo. Comme on l'a vu au [Chapitre 17](#), un algorithme de Monte Carlo consiste à effectuer des sélections aléatoires au cours d'un processus d'optimisation, sans offrir la garantie d'un succès, mais avec une probabilité élevée de mener à bien la tâche. Le problème n'est cependant pas purement théorique, sachant que cette méthode sert notamment à tester le bon fonctionnement des circuits électroniques et à les optimiser en supprimant les parties de circuit qui ne transmettent pas de signaux électriques. Par ailleurs, l'algorithme de résolution trouve d'autres applications : étiquetage automatique sur des cartes et des graphiques, tomographie discrète, planification de tâche sous contraintes, regroupement de données par grappes et autres problèmes impliquant des choix contradictoires.

Les circuits des ordinateurs sont constitués d'une série de composants connectés, chaque composant ouvrant ou fermant un circuit en fonction de ses inputs. Ces éléments sont ce que l'on appelle des *portes logiques* (physiquement parlant, ce rôle est assuré par des transistors) et pour assembler un circuit pouvant comporter un certain nombre de portes logiques, il importe de savoir si le courant électrique peut les traverser, et si oui, dans quelles circonstances.

Le [Chapitre 14](#) traite de la représentation interne d'un ordinateur, sur la base des 0 (absence de courant électrique dans le circuit) et des 1 (présence de courant électrique). Cette représentation binaire peut être envisagée d'un point de vue logique en distinguant deux

conditions possibles d'un signal, Faux (pas de courant dans le circuit) et Vrai (du courant). Le [Chapitre 4](#) présente les opérateurs booléens (AND, OR et NOT), comme le rappelle la [Figure 18-4](#), les conditions True (Vrai) et False (Faux) étant les inputs, transformés pour donner un résultat différent. Tous ces concepts vous permettent de représenter un circuit électrique physique sous la forme d'une séquence d'opérateurs booléens qui définissent des portes logiques. La combinaison de toutes les conditions détermine si le circuit peut transmettre un courant électrique.

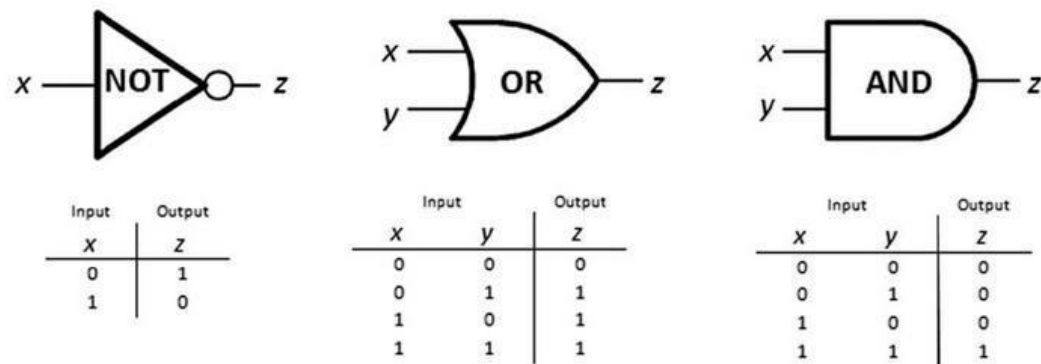


FIGURE 18-4 Symboles et tables de vérité des opérateurs logiques AND, OR et NOT.

Cette représentation logique est un *circuit combinatoire booléen*, et le test de vérification de sa fonctionnalité est la *satisfiabilité du circuit*. Dans le meilleur des cas, le circuit ne comporte que des conditions NOT (appelées des *inverseurs*) acceptant une entrée à un fil, et des conditions OR acceptant une entrée à deux fils. Il s'agit de la situation de satisfiabilité limitée à deux littéraux (2-SAT), et si l'algorithme devait inclure cette séquence dans le cadre d'une recherche exhaustive, il faudrait dans le pire des cas 2^k essais (k étant le nombre de fils entrants) pour trouver un ensemble de conditions permettant la circulation du courant électrique à travers l'ensemble du circuit. Il existe des versions encore plus complexes de ce problème, avec davantage d'entrées pour chaque porte logique OR et avec des portes AND, mais elles sortent du cadre de ce livre.

Résoudre 2-SAT grâce à la randomisation

Quel que soit le circuit électronique à tester, vous pouvez le représenter sous la forme d'un vecteur de variables booléennes. Vous pouvez aussi créer un autre vecteur qui contiendra les *clauses*, c'est-à-dire les conditions auxquelles le circuit doit satisfaire (par exemple, que les entrées A et B aient l'une et l'autre le statut True). Ce n'est pas la seule façon de représenter le problème : il existe d'autres possibilités, qui font appel aux graphes. Cependant, dans le cadre de cet exemple, ces deux vecteurs suffisent.

La résolution du problème se fait sous la forme d'une recherche locale randomisée en temps polynomial. L'algorithme en question, *RandomWalkSAT*, a été présenté par Christos H. Papadimitriou, professeur à l'Université de Californie de Berkeley (<https://people.eecs.berkeley.edu/~christos/>), dans son article « On Selecting a Satisfying Truth Assignment » publié en 1991 parmi les comptes-rendus du 32^e colloque international de l'IEEE sur les fondements de l'informatique. Cet algorithme tient la comparaison avec les méthodes plus rationnelles, et il s'agit d'une méthode de recherche locale exemplaire, car la solution courante ne fait l'objet que d'un changement à la fois. Il utilise deux boucles imbriquées, une pour tester plusieurs fois la solution de départ et une pour modifier de façon aléatoire la solution aléatoire initiale. La boucle externe est répétée $\log_2(k)$ fois (k étant le nombre d'entrées). La boucle interne effectue les étapes suivantes :

1. **Sélection d'une solution au hasard.**
2. **Répétition des étapes suivantes $2 * k^2$ fois :**
 - a. Détermine si la solution courante est la bonne. Si oui, sortie de toutes les boucles et affichage de cette solution.

b. Sélectionne au hasard une clause non satisfaite. Choisit au hasard une des conditions qu'elle comporte et la modifie.

Implémenter le code Python

Pour résoudre le problème 2-SAT à l'aide de Python et de l'algorithme *RandomWalkSAT*, il faut une petite série de fonctions. Les fonctions `create_clauses` et `signed` permettent de formaliser un problème de circuit à résoudre en gérant respectivement des portes OR et des portes NOT. En utilisant ces fonctions, vous spécifiez le nombre de portes OR et déterminez une valeur de départ qui garantissent la possibilité de reproduire par la suite le problème résultant (vous pourrez résoudre ce problème plusieurs fois sur différents ordinateurs).

La fonction `create_random_solutions` donne un point de départ en produisant une solution aléatoire qui détermine les inputs. Les chances de trouver la bonne solution par hasard sont minces (une chance sur deux puissance le nombre de portes), mais en moyenne on peut s'attendre à ce que les trois quarts des portes soient correctement initialisées (en effet, comme on l'a vu en utilisant la table de vérité pour la fonction OR, trois entrées sur les quatre possibles ont le statut `True`). La fonction `check_solution` détermine le moment où les conditions du circuit sont satisfaites (ce qui indique une solution correcte). Autrement, elle énonce les conditions qui ne sont pas satisfaites (vous pouvez retrouver ce code dans le fichier téléchargeable A4D ; 18 ; Local Search.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import numpy as np
import random
from math import log2

import matplotlib.pyplot as plt
%matplotlib inline

def signed(v):
    return v if np.random.random()<0.5 else -v
```

```

def create_clauses(i, seed=1):
    np.random.seed(seed)
    return [(signed(np.random.randint(i)), signed(
        np.random.randint(i))) for j in range(i)]

def create_random_solution(i, *kwargs):
    return {j:signed(1)==1 for j in range(i)}

def check_solution(solution, clauses):
    violations = list()
    for k,(a,b) in enumerate(clauses):
        if not (((solution[abs(a)]) == (a>0)) |
            ((solution[abs(b)]) == (b>0))):
            violations.append(k)
    return violations

```

Après avoir établi ces fonctions, vous disposez de tous les éléments nécessaires à la définition d'une fonction `sat2` pour résoudre le problème. Cette solution utilise deux itérations imbriquées : la première reproduit un certain nombre de départs ; la seconde sélectionne les conditions non satisfaites de façon aléatoire et les rend vraies. La solution est obtenue dans un temps polynomial. La fonction ne présente pas la garantie de trouver une solution, si elle existe, mais elle a de bonnes chances de la trouver. En effet, la boucle interne effectue $2 * k^2$ tentatives aléatoires de résoudre le problème de circuit, ce qui est généralement suffisant pour qu'une marche aléatoire sur une ligne atteigne sa destination.



Une *marche aléatoire* est une série de computations représentant un objet qui s'éloigne de sa position initiale en prenant à chaque étape une direction au hasard. On pourrait imaginer la trajectoire d'un individu ivre d'un réverbère au réverbère suivant. Les marches aléatoires sont utiles pour représenter un modèle mathématique reflétant un certain nombre d'aspects du monde réel. Elles trouvent des applications en biologie, en physique, en chimie, en informatique et en économie, en particulier dans l'analyse du marché boursier. Pour en savoir davantage sur les marches aléatoires, consultez la page [http://www.mit.edu/~kardar/teaching/projects/chemotaxis\(AndreaSchmidt\)/random.htm](http://www.mit.edu/~kardar/teaching/projects/chemotaxis(AndreaSchmidt)/random.htm).

Une marche aléatoire sur une ligne est l'exemple de marche aléatoire le plus simple. En moyenne, k^2 étapes sont nécessaires pour arriver à une distance k du point de départ. Compte tenu de cette espérance d'effort, RandomWalkSAT a $2 * k^2$ chances de modifier la solution de départ. Ce nombre de chances correspond à une probabilité élevée que l'algorithme résolve les k clauses. Le principe est le même que pour le jeu consistant à deviner une carte tirée d'un paquet, dont il était question au chapitre précédent. À mesure que l'algorithme se déroule, le choix de la bonne réponse devient plus facile. Les répliques externes garantissent la possibilité d'échapper aux choix restreints de la boucle interne qui pourraient bloquer le processus dans une solution locale.

```
def sat2(clauses, n, start=create_random_solution):
    for external_loop in range(round(log2(n))):
        solution = start(n, clauses)
        history = list()
        for internal_loop in range(2*n**2):
            response = check_solution(solution, clauses)
            unsatisfied = len(response)
            history.append(unsatisfied)
            if unsatisfied==0:
                print ("Solution trouvée avec %i boucles
externes," %
                    (external_loop+1), end=" ")
                print ("%i boucles internes" %
                    (internal_loop+1))
                break
            else:
                r1 = random.choice(response)
                r2 = np.random.randint(2)
                clause_to_fix = clauses[r1][r2]
                solution[abs(clause_to_fix)] = (
                    clause_to_fix>0)
        else:
            continue
        break
    return history, solution
```

Maintenant que toutes les fonctions sont établies correctement, vous pouvez exécuter le code pour résoudre un problème. Voici le

premier exemple, consistant à tester le circuit créé à la position 0 avec 1 000 portes logiques.

```
n = 1000
# Valeurs de départ permettant une résolution avec
n=1000 :
0,1,2,3,4,5,6,9,10
# Valeurs de départ ne permettant pas une résolution
avec n=1000 : 8
clauses = create_clauses(n, seed=0)
history, solution = sat2(clauses, n,
                        start=create_random_solution)
```

Solution trouvée avec 1 boucle externe\$, 1360 boucles internes

À l'aide d'un graphique représentant le nombre d'étapes en abscisse (améliorations aléatoires de la solution) et les clauses qu'il reste à résoudre en ordonnées, vous pouvez vérifier que l'algorithme finit généralement par trouver la solution correcte ([Figure 18-5](#)).

```
plt.plot(np.array(history), 'b-')
plt.xlabel("Ajustements aléatoires")
plt.ylabel("Clauses non satisfaites")
plt.grid(True)
plt.show()
```

Si vous testez le circuit avec 1 000 portes et une valeur de départ égale à 8, vous remarquerez que le processus semble ne jamais devoir finir. En effet, le problème du circuit ne peut pas être résolu, et tous ces choix aléatoires et ces tentatives demandent un temps long. À la fin, l'algorithme ne vous donnera aucune solution.

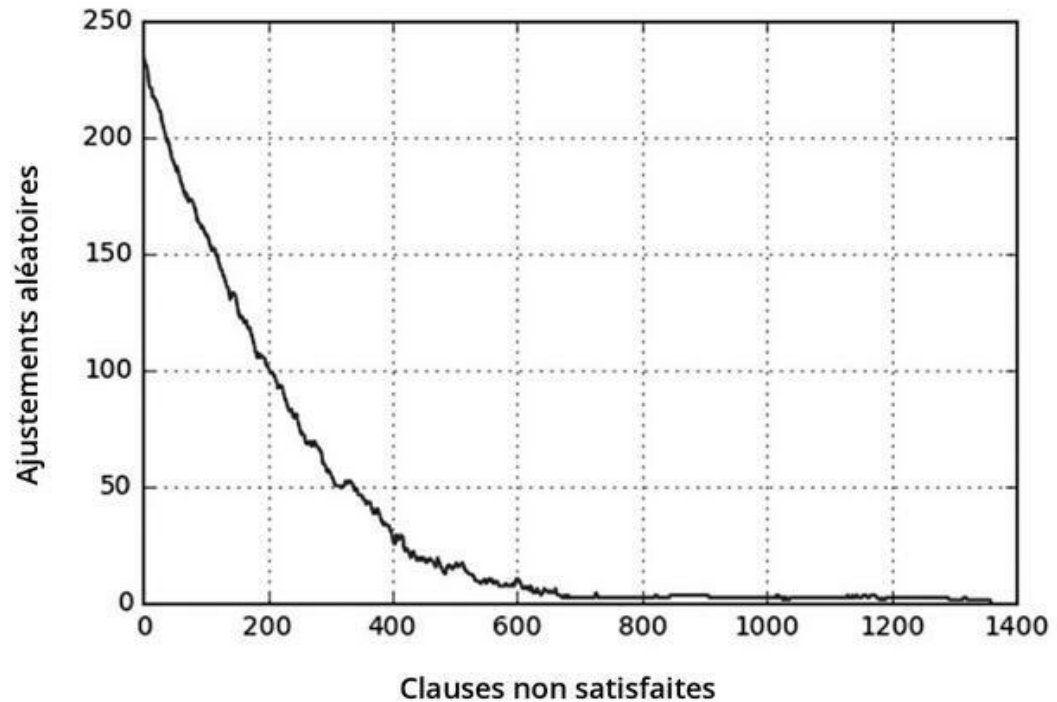


FIGURE 18-5 Le nombre de clauses non satisfaites décroît après les ajustements aléatoires.

Réaliser l'importance du point de départ

Même si l'algorithme RandomWalkSAT présente une complexité d'exécution de $O(\log 2k * k^2)$ dans le pire des cas, k étant le nombre d'entrées, il est possible de le rendre plus rapide en hachant le point de départ. En effet, bien qu'une configuration aléatoire au départ implique qu'en moyenne, une clause sur quatre reste non satisfaite au départ, on peut en résoudre un certain nombre en passant en revue les données.

Le problème des clauses est qu'un certain nombre d'entre elles exigent un input vrai, tandis qu'en même temps, un certain nombre d'autres exigent un input faux. Quand toutes les clauses exigent qu'un input soit vrai ou qu'il soit faux, on peut le fixer de manière à satisfaire cette condition, si bien qu'un grand nombre de

clauses sont satisfaites et la résolution de celles qui restent est rendue plus facile. La nouvelle application de RandomWalkSAT qui suit comporte une phase de départ qui résout immédiatement les situations dans lesquelles une entrée doit impérativement avoir le statut vrai ou le statut faux selon toutes les clauses qui les concernent :

```
def better_start(n, clauses):
    clause_dict = dict()
    for pair in clauses:
        for clause in pair:
            if abs(clause) in clause_dict:
                clause_dict[abs(clause)].add(clause)
            else:
                clause_dict[abs(clause)] = {clause}

    solution = create_random_solution(n)

    for clause, value in clause_dict.items():
        if len(value)==1:
            solution[clause] = value.pop() > 0
    return solution
```

Ce code définit une nouvelle fonction pour le démarrage à froid. Après avoir généré une solution aléatoire, il examine la solution et trouve toutes les entrées associées à un état unique (vrai ou faux). En les initialisant immédiatement à l'état exigé, on peut réduire le nombre de clauses impliquant un ajustement. Ainsi, la recherche locale comporte moins de tâches et se termine plus tôt.

```
n = 1000
# Valeurs de départ permettant une résolution =
0,1,2,3,4,5,6,9,10
# Valeurs de départ ne permettant pas une résolution =
8
clauses = create_clauses(n, seed=0)
history, solution = sat2(clauses, n,
start=better_start)

Solution trouvée avec 1 boucle externe, 393 boucles
internes
```

Avec ce nouveau point de départ simplifié, après avoir représenté les résultats, vous pouvez immédiatement constater un progrès sachant qu'en moyenne, moins d'opérations sont nécessaires pour exécuter la tâche.

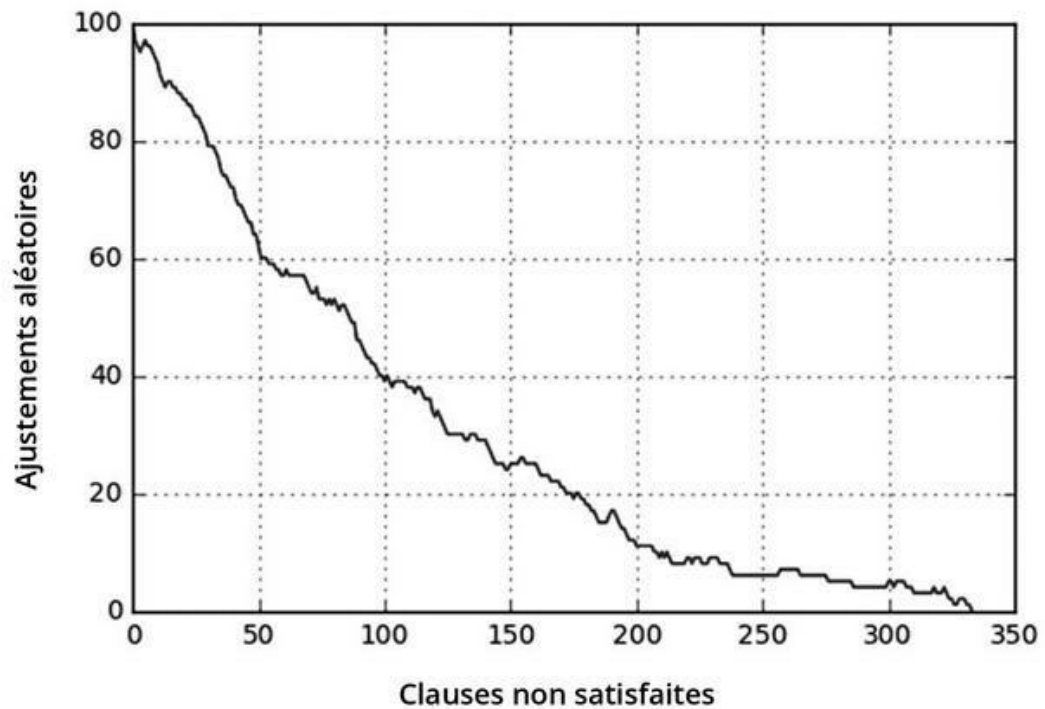


FIGURE 18-6 L'exécution est plus rapide grâce à un meilleur point de départ.



Dans une recherche locale, il ne faut jamais sous-estimer l'importance du point de départ pour permettre que l'algorithme termine sa tâche plus rapidement et avec plus de succès ([Figure 18-6](#)). En résumé, pour toute recherche, il convient de réunir les meilleures conditions de départ possibles.

Chapitre 19

Faire appel à la programmation linéaire

DANS CE CHAPITRE

- » Découvrir le rôle de la programmation linéaire dans l'optimisation
 - » Transformer des problèmes du monde réel en problèmes de mathématiques et de géométrie
 - » Apprendre à utiliser Python pour résoudre des problèmes de programmation linéaire
-

La programmation linéaire a fait sa première apparition durant la Seconde Guerre mondiale, alors que la logistique se révélait essentielle pour faire évoluer des armées de millions de soldats, des armes et du ravitaillement sur des champs de bataille géographiquement hétérogènes. Il fallait réapprovisionner en carburant et en munitions les tanks et les avions, ce qui demandait des efforts d'organisation considérables dans un contexte de limites de temps, de ressources limitées et d'actions hostiles de la part de l'ennemi.

De tels problèmes militaires, la plupart du temps, peuvent être exprimés sous une forme mathématique. Nous devons au mathématicien George Bernard Dantzig, qui travaillait à l'Office de Contrôle statistique de l'U.S. Air Force, une méthode judicieuse de résolution de ces problèmes par l'algorithme du *simplex*. Le *simplex* est l'idée de base qui a suscité l'intérêt pour

l'optimisation numérique après la guerre et qui a donné naissance à cette discipline prometteuse qu'est la programmation linéaire. L'avènement des premiers ordinateurs performants, à cette époque, a également suscité l'intérêt et a permis d'exécuter des calculs complexes d'une nouvelle manière, bien plus rapide. L'histoire des premiers temps de l'informatique, dans les années cinquante et soixante, peut être considérée comme la recherche d'une optimisation des problèmes de logistique par la méthode du simplex, avec le recours simultané à des ordinateurs rapides et à des langages de programmation spécialisés.

Dantzig est décédé en 2005, et le domaine d'étude dont il est à l'origine est toujours en continuel développement. Ces dernières années, dans le domaine de la programmation linéaire, de nouvelles idées et de nouvelles méthodes ont connu le succès :

- » **La programmation par contraintes** : elle exprime les relations entre les variables dans un programme informatique comme des contraintes en programmation linéaire.
- » **Les algorithmes génétiques** : ils sont fondés sur l'idée que des formules mathématiques peuvent se reproduire et muter pour résoudre des problèmes de la même manière que l'ADN le fait dans la nature, par l'évolution. Les algorithmes génétiques sont aussi abordés au [Chapitre 20](#), en raison de leur approche heuristique de l'optimisation.

Ce chapitre vous aide à comprendre la programmation linéaire. Il vous montre également comment l'appliquer à des problèmes du monde réel en utilisant Python comme outil pour formaliser ces problèmes sous forme de code.

Utiliser les fonctions linéaires comme outil

Cette section explique comment traiter un problème dans lequel l'*objectif* (la représentation du coût, du profit ou d'une autre grandeur à maximiser ou à minimiser en tenant compte des contraintes) et les *contraintes* du problème (des inégalités linéaires tirées de l'application, comme une limite de 40 heures à l'horaire de travail hebdomadaire) sont formalisés sous forme de fonctions linéaires. L'objet de la programmation linéaire est de déterminer une solution numérique optimale, pouvant être une valeur maximum ou minimum, avec l'ensemble des conditions pour l'obtenir.

Cette définition peut sembler un peu compliquée sachant qu'elle fait référence à des notions de mathématiques et à un certain degré d'abstraction (les objectifs et les contraintes étant mis sous forme de fonctions linéaires), mais les choses deviennent plus claires quand on maîtrise la notion de fonction et quand on sait déterminer si une fonction est ou n'est pas linéaire. Au-delà du jargon mathématique, la programmation linéaire est simplement un point de vue différent pour aborder les problèmes d'algorithmique, consistant à considérer non plus des opérations et des manipulations de données entrées, mais des fonctions mathématiques, et à effectuer des calculs en utilisant un *logiciel d'optimisation* ou *optimiseur*.

La programmation linéaire ne permet pas de résoudre tous les problèmes, mais un grand nombre de problèmes répondent aux exigences de cette méthode, en particulier les problèmes d'optimisation dans lesquels il faut tenir compte de limites préalablement définies. Les chapitres précédents étudient dans quelle mesure la programmation dynamique est la meilleure méthode d'optimisation lorsqu'un problème comporte des contraintes. La programmation dynamique convient pour résoudre des problèmes à variable discrète, c'est-à-dire dans lesquels les nombres avec lesquels on travaille sont des nombres entiers. Au contraire, la programmation linéaire traite généralement des nombres décimaux, encore qu'il existe des algorithmes d'optimisation spéciaux donnant des solutions sous forme de nombres entiers (le problème du voyageur de commerce, par

exemple, peut être résolu par une programmation linéaire de nombres entiers). Le champ d'application de la programmation linéaire est plus large, sachant qu'elle peut être appliquée pour pratiquement tout problème résoluble en temps polynomial.

La programmation linéaire trouve des applications dans l'industrie, la logistique, les transports (en particulier dans les compagnies aériennes, pour la détermination des itinéraires, des horaires et du prix des billets), le marketing, la finance et les télécommunications. Dans toutes ces applications, il s'agit de maximiser un résultat économique et de minimiser un coût tout en optimisant l'allocation des ressources disponibles et tout en respectant toutes les contraintes et toutes les limitations. Par ailleurs, la programmation linéaire peut être utilisée aussi dans des applications courantes comme les jeux vidéo et la visualisation sur ordinateur, sachant que les jeux font appel à la gestion de formes bidimensionnelles et tridimensionnelles complexes et qu'il faut gérer les éventuelles collisions entre ces formes et veiller au respect des règles du jeu. Pour atteindre ces objectifs, on doit recourir au calcul de l'enveloppe convexe, par le moyen de la programmation linéaire (voir http://www.tcs.fudan.edu.cn/rudolf/Courses/Algorithms/Alg_ss_07). Enfin, la programmation linéaire intervient dans les moteurs de recherche, pour les problèmes de récupération de documents : on peut transformer en fonctions les mots, les expressions et les documents et déterminer une méthode de maximisation du résultat des recherches (obtenir les documents nécessaires pour répondre à la requête), quand on recherche des documents présentant certaines caractéristiques mathématiques.

Acquérir les bases mathématiques nécessaires

En programmation, une fonction permet de disposer d'une série de lignes de code que l'on compte utiliser plus d'une fois. Avec les fonctions, le code devient une boîte noire, c'est-à-dire une

entité qui doit produire certains outputs à partir des inputs dont elle est alimentée. Le [Chapitre 4](#) explique comment créer des fonctions dans Python. En mathématiques, on utilise des fonctions de façon similaire pour programmer : les fonctions sont des séries d'opérations mathématiques qui transforment des inputs en outputs. L'input peut être constitué d'une ou plusieurs variables, à partir desquelles la fonction produit un output unique. Une fonction a généralement la forme suivante :

$$f(x) = x * 2$$

- » **f** est le nom de la fonction dans cet exemple, mais ce nom peut être n'importe quelle lettre de l'alphabet, ou même un mot.
- » **(x)** spécifie l'input. Dans cet exemple, l'input est la variable x, mais on peut utiliser davantage d'inputs. Dans des cas plus complexes, l'input peut être constitué de plusieurs variables ou d'une ou plusieurs matrices.
- » **x * 2** définit la série d'opérations exécutées par la fonction sur l'input qui lui est transmis. Le résultat de la fonction est un nombre.

Dans cet exemple, si la valeur 2 est attribuée à x, on obtient :

$$f(2) = 4$$

En termes mathématiques, on dira que nous avons appelé cette fonction avec 2 en entrée et que nous avons obtenu 4 en sortie.



Une fonction peut être simple ou complexe, mais toute fonction donne un résultat et un seul pour toute série d'inputs utilisée (même si l'input est constitué de plusieurs variables).

La programmation linéaire utilise des fonctions pour formaliser mathématiquement les objectifs à atteindre, en vue de la résolution du problème concerné. Quand les objectifs sont présentés sous la forme d'une fonction mathématique, le problème consiste à déterminer l'input de la fonction qui permettra d'obtenir

un output maximum (ou minimum, selon ce qu'il s'agit de réaliser). La fonction qui représente l'objectif d'optimisation est la fonction objectif. Par ailleurs, la programmation linéaire utilise des fonctions et des inégalités pour exprimer les contraintes ou les limites qui empêchent d'envisager n'importe quel input pour la fonction objectif. Voici deux inégalités :

$$\begin{aligned}0 &\leq x \leq 4 \\ y + x &< 10\end{aligned}$$

La première de ces inégalités signifie que l'input de la fonction objectif se limite à des valeurs comprises entre 0 et 4. Une inégalité peut aussi faire intervenir plus d'une variable d'entrée à la fois. Dans cet exemple, la seconde inégalité établit un lien entre les valeurs d'une entrée et celles d'une autre : leur somme doit être inférieure à 10.



Une *limite* restreint l'étendue des valeurs possibles d'un input, comme dans le premier exemple, tandis qu'une *contrainte* est une expression mathématique faisant intervenir plus d'une variable, comme dans le second exemple.

Une dernière exigence de la programmation linéaire est que la fonction objectif et les inégalités soient des expressions linéaires. Cela signifie qu'elles ne peuvent pas comporter des variables qui seraient multipliées l'une par l'autre, ni élevées à une certaine puissance (au carré ou au cube, par exemple).



Dans le cadre d'une optimisation, toutes les fonctions doivent être des expressions linéaires, car la procédure les représente sous forme de droites dans un espace cartésien (pour un rappel de la notion d'espace cartésien, consultez la page <http://www.mathsisfun.com/data/cartesian-coordinates.html>).

Comme cela est expliqué dans la section « Faire une application pratique de la programmation linéaire », plus loin dans ce chapitre, l'utilisation de la programmation linéaire peut s'envisager comme la résolution d'un problème de géométrie plutôt que d'algèbre ou d'analyse.

Apprendre à simplifier une planification

Les problèmes qui étaient résolus par l'algorithme du simplexe dans sa version originale relevaient tous du type de problème qu'on a l'habitude de trouver dans un ouvrage de mathématiques. Dans ces problèmes, toutes les données, toutes les informations et toutes les limitations sont énoncées clairement, il n'y a pas d'information redondante ou non pertinente, et il s'agit évidemment d'appliquer une formule mathématique (celle qui vient d'être étudiée) pour résoudre le problème.

Dans la pratique, les solutions n'apparaissent jamais aussi claires. Souvent, elles sont plutôt confuses et il manque certaines informations importantes. On peut cependant analyser le problème et trouver les données et autres informations nécessaires. Par ailleurs, il peut y avoir des limitations à prendre en compte en termes de ressources financières ou de temps, ou bien des règles ou un ordre à respecter. En vue de résoudre un problème, il faut rassembler l'information et trouver un moyen de la simplifier.

Une simplification implique une certaine perte, mais elle rend les choses plus faciles à gérer. On peut ainsi mettre en évidence les processus par lesquels une situation évolue, en vue d'aider la décision. Simplifier un problème permet de développer un modèle représentant la réalité de façon approximative, et ce modèle peut servir à des simulations ou à une programmation linéaire.

Supposons que vous travailliez dans une usine et que vous deviez planifier un programme de production en sachant que plus vous affecterez de personnes à ce projet, plus la production sera rapide. Cependant, l'ajout d'une même quantité de personnel n'entraînera pas toujours le même gain. Les résultats dépendront, par exemple, des compétences des personnes affectées au projet. Par ailleurs, vous pourrez constater que lorsque vous ajoutez du personnel, la performance se dégrade car les gens consacrent davantage de temps à communiquer et à se coordonner qu'à faire du travail

utile. Vous pouvez cependant simplifier le modèle en faisant comme si chaque personne affectée à la tâche allait produire une certaine quantité de produits finis ou de biens intermédiaires.

Faire de la géométrie avec le simplex

Les exemples classiques de programmation linéaire concernent une production de biens avec des ressources limitées (en termes de temps, de main-d'œuvre ou de matériaux). À titre d'exemple, imaginons une unité de production qui assemblerait deux produits x et y , les journées de travail étant de 8 heures, et qui devrait les livrer dans un certain délai. Le profit tiré de chaque unité produite serait différent selon le produit (ce profit étant calculé en soustrayant les coûts des recettes), de même que les taux de production horaires et la demande journalière sur le marché :

- » Recettes en euros pour chaque produit : $x = 15$, $y = 25$
- » Taux de production horaire : $x = 50$, $y = 40$
- » Demande journalière par produit : $x = 300$, $y = 200$

En substance, le problème est de décider s'il convient de produire davantage de x , plus faciles à assembler mais moins rémunérateurs, ou de y , un produit qui garantit des recettes plus importantes mais dont la production demande plus de temps. Pour résoudre ce problème, il faut tout d'abord déterminer la fonction objectif. Elle s'exprime comme la somme des quantités des deux produits multipliée par leur espérance de recette unitaire, et vous savez qu'il s'agit de la maximiser (on ne doit minimiser la fonction objectif que lorsqu'il s'agit d'un problème de coût) :

$$f(x, y) = 15 * x + 25 * y$$

Ce problème comporte des inégalités qui déterminent les valeurs possibles de x et de y telles que l'optimisation puisse donner un résultat valide :

$$0 \leq x \leq 300$$

$$0 \leq y \leq 200$$

En effet, on ne peut pas produire un nombre négatif d'unités, d'autre part, il ne serait pas justifié d'en produire plus que ce que le marché demande. Une autre limitation importante est le temps disponible, sachant qu'une journée de travail ne peut pas dépasser 8 heures. Il faut donc calculer le temps nécessaire pour produire x et y et limiter le temps total à 8 heures :

$$x/40 + y/50 \leq 8$$

Vous pouvez représenter les fonctions sur un plan cartésien (pour un rappel sur la représentation graphique des fonctions, consultez la page <http://www.mathpla-net.com/education/pre-algebra/graphingand-functions/linear-equations-in-the-coordinate-plane>). Sachant que dans ce problème, tout peut être exprimé par des fonctions, la programmation linéaire peut prendre la forme d'un problème géométrique dans un espace de coordonnées cartésiennes, et s'il n'y a pas plus de deux variables, il est possible de représenter les deux fonctions et leurs contraintes sous forme de droites sur un plan et de déterminer le polygone que ces droites délimitent. Ce polygone est la *zone d'acceptabilité*, celle à l'intérieur de laquelle se trouve la solution et qui contient tous les inputs valides (compte tenu des contraintes).



Quand le problème met en jeu plus de deux variables, on peut encore envisager d'étudier des intersections de droites dans un espace, mais il n'est plus possible d'en faire une représentation sur papier, car à chaque variable d'entrée correspond une dimension sur le graphique et il n'est pas envisageable d'aller au-delà des trois dimensions du monde dans lequel nous vivons.

À ce stade, l'algorithme de programmation linéaire explore la zone d'acceptabilité et indique la solution. Pour déterminer la meilleure solution du problème, il n'est pas nécessaire de passer en revue chaque point de la zone délimitée. Imaginons que la fonction objectif soit une autre droite représentée sur le plan (sachant que la fonction objectif est aussi une fonction linéaire). Vous pouvez constater que la solution recherchée est constituée

des points d'intersection entre la zone d'acceptabilité et la droite de la fonction objectif ([voir Figure 19-1](#)). Si l'on parcourt la droite de la fonction objectif dans le sens de la descente, on atteint à un moment donné un point de la zone d'acceptabilité. Ce point de contact est généralement un des sommets de cette zone, mais il peut aussi s'agir d'un côté tout entier du polygone (auquel cas chaque point de ce segment de droite est une solution optimale).

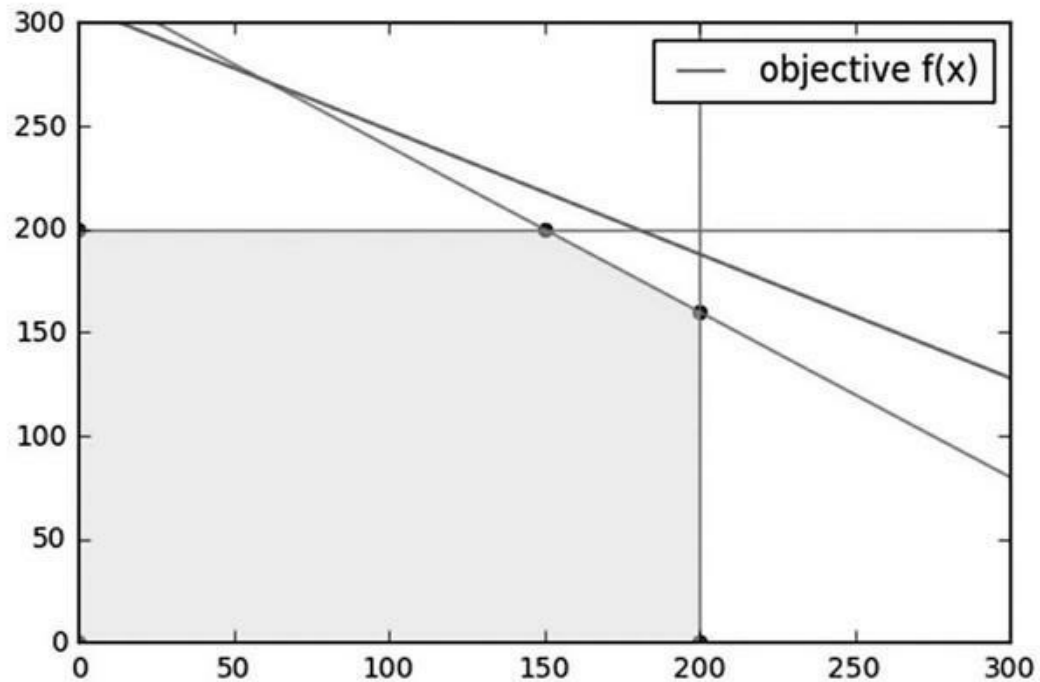


FIGURE 19-1 Recherche de l'intersection entre la fonction objectif et la zone d'acceptabilité.

Du point de vue pratique, l'algorithme du simplexe ne peut pas faire apparaître des droites descendantes comme dans cet exemple. Il parcourt plutôt la bordure de la zone d'acceptabilité (en dénombrant les sommets) et teste les valeurs de la fonction objectif à chaque sommet jusqu'à ce qu'il ait déterminé la solution. Le temps réel d'exécution dépend donc du nombre de sommets, lequel dépend du nombre de contraintes et de variables intervenant dans la détermination de la solution (plus de variables signifie plus de dimensions et plus de sommets).

Faire le point des limitations

À mesure que vous maîtrisez mieux la programmation linéaire et que les problèmes deviennent plus difficiles, vous êtes amené à utiliser des méthodes plus avancées que la version de base de l'algorithme du simplex présentée dans ce chapitre. En fait, le simplex n'est plus utilisé car des algorithmes plus élaborés l'ont remplacé : des algorithmes qui traversent la zone d'acceptabilité au lieu d'en parcourir la périphérie. Ces algorithmes trouvent un raccourci là où le simplex rechercherait la solution du mauvais côté de cette zone.

Travailler avec des nombres à virgule flottante est parfois contraignant aussi, sachant que de nombreux problèmes appellent une réponse binaire (1/0) ou entière. Par ailleurs, d'autres problèmes nécessitent une représentation à l'aide de courbes plutôt que de droites. Certains logiciels du marché mettent en œuvre des algorithmes de programmation linéaire entière ou de programmation non linéaire. Sachez simplement que les problèmes de programmation linéaire entière et de programmation non linéaire sont des problèmes NP-complets et que leur résolution peut exiger autant de temps, sinon davantage, que les autres algorithmes que vous connaissez.

Faire une application pratique de la programmation linéaire

Le meilleur moyen de s'initier à la programmation linéaire est d'utiliser des solutions prédéfinies plutôt que de créer soi-même des applications *ad hoc*. La prochaine section vous explique comment installer une solution prédéfinie pour les exemples qui suivent.



Si vous travaillez sur ordinateur, vous remarquerez peut-être des différences significatives entre les logiciels libres et les produits vendus dans le commerce. Les premiers offrent un vaste ensemble

d'algorithmes, mais dont l'exécution peut décevoir lorsque les problèmes sont vastes ou complexes. Intégrer des algorithmes de programmation linéaire dans des logiciels est compliqué, et l'on ne peut pas s'attendre à ce qu'un logiciel libre soit aussi performant et aussi parfait qu'un logiciel vendu dans le commerce.

Les logiciels en source ouverte offrent tout de même des possibilités intéressantes d'apprentissage. Les sections qui suivent utilisent une solution Python en source ouverte appelée PuLP, qui vous permet de développer des optimisations par programmation linéaire après avoir défini une fonction de coût et des contraintes sous forme de fonctions Python. Il s'agit essentiellement d'une solution didactique, qui convient pour découvrir l'utilisation de la programmation linéaire et la formulation des problèmes en termes mathématiques.



PuLP offre une interface pour les programmes solveurs et vous permet d'accéder au programme solveur par défaut en source ouverte de Python. La performance (rapidité, pertinence et adaptabilité) de PuLP dépend presque entièrement du solveur et de l'optimiseur que l'utilisateur a choisis. Les meilleurs solveurs sont des produits commerciaux comme CPLEX (<https://fr.wikipedia.org/wiki/CPLEX>), Xpress (https://en.wikipedia.org/wiki/FICO_Xpress) et GuRoBi (<https://en.wikipedia.org/wiki/Gurobi>), qui présentent un avantage considérable en termes de rapidité par rapport aux solveurs en source ouverte.

Installer PuLP chez soi

PuLP est un projet Python en source ouverte créé par Jean-Sébastien Roy. Par la suite, il a été modifié et tenu à jour par Stuart Antony Mitchell. Le module PuLP vous permet de définir des problèmes de programmation linéaire et de les résoudre en utilisant le solveur intégré (qui repose sur l'algorithme du simplex). Vous pouvez aussi utiliser d'autres solveurs du domaine

public ou disponibles moyennant paiement d'une licence. Le projet (contenant tout le code source et de nombreux exemples) se trouve à l'adresse <https://github.com/coin-or/pulp>. La documentation complète se trouve à l'adresse <https://pythonhosted.org/PuLP/>.

PuLP n'étant pas disponible dans Anaconda, vous devez l'installer vous-même. Pour ce faire, vous devez utiliser la version Anaconda3 (ou plus récente) de l'invite de commande, car les versions plus anciennes ne fonctionneraient pas. Ouvrez une interface de commande, tapez **pip install pulp** et appuyez sur la touche Entrée. Si vous avez accès à l'Internet, la commande pip déclenchera le téléchargement du module PuLP et son installation dans Python (la version utilisée dans les exemples de ce chapitre est PuLP 1.6.1, mais les versions ultérieures devraient offrir les mêmes fonctionnalités).

Optimiser la production et les recettes

Cette section étudie un autre problème d'optimisation dans le domaine de la production. Ce problème concerne deux produits A et B (et met donc en jeu deux variables que vous pourrez représenter sur un graphique à deux dimensions). Le processus de production est constitué d'une série de transformations en trois phases. Chaque phase fait appel à un certain nombre d'opérateurs (la valeur n) qui peuvent être des ouvriers ou des robots, et les ressources nécessaires à chaque phase ne sont opérationnelles que pendant un certain nombre de jours par mois (représenté par la valeur t). Le processus subi par un produit est différent à chaque phase, et dure un certain nombre de jours. Dans la première phase (appelée « res_1 »), par exemple, un opérateur met deux jours pour terminer le produit A, mais trois jours pour le produit B. Enfin, le profit unitaire n'est pas le même sur chaque produit : le produit A rapporte 3 000 euros l'unité et le produit B 2 500 euros. Le tableau suivant résume le problème :

Étape de production	Temps pour le produit A par opérateur (en jours)	Temps pour le produit B par opérateur (en jours)	Disponibilité (en jours)	Nb opérateurs
res_1	2	3	30	2
res_2	3	2	30	2
res_3	3	3	22	3

Pour trouver la fonction objectif, calculez la somme des quantités des deux produits multipliées par les profits unitaires respectifs. Il s'agit de maximiser ce résultat. Il existe des contraintes, même si elles ne sont pas énoncées de façon explicite. Tout d'abord, à chaque phase, une contrainte de disponibilité limite la productivité. La deuxième contrainte est le nombre d'opérateurs. La troisième est la productivité relative des deux produits. Le problème peut être reformulé en faisant référence au temps utilisé pour fabriquer chaque produit à chaque phase, qui ne peut dépasser le temps disponible multiplié par le nombre d'opérateurs disponibles. Le nombre d'opérateurs multiplié par le nombre de jours ouvrés donne les ressources en temps utilisables. Ces ressources ne peuvent pas être inférieures au temps nécessaire pour produire tous les produits qu'il s'agit de livrer. Les formulations qui en résultent sont les suivantes, avec les contraintes pour chaque phase :

```

objective = 3000 * qty_A + 2500 * qty_B
           production_rate_A * qty_A + production_rate_B *
           qty_B
           <= uptime_days * workers

```

Chaque contrainte peut être exprimée en faisant dépendre la quantité d'un produit de la quantité de l'autre (en effet, si l'on produit A et si la production de A consomme tout le temps disponible, il n'est plus possible de produire B) :

```
qty_B <= ((uptime_days * workers) -
          (production_rate_A * qty_A) ) / production_rate_B
```

Vous pouvez enregistrer toutes les valeurs relatives à chaque phase pour production_rate_A, production_rate_B, uptime_days et workers afin d'accéder plus facilement à un dictionnaire de Python. Les profits doivent rester dans les variables (vous retrouverez ce code dans le fichier téléchargeable A4D ; 19 ; Linear Programming.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import numpy as np
import matplotlib.pyplot as plt
import pulp

%matplotlib inline

res_1 = {'A':2, 'B':3, 't':30, 'n':2}
res_2 = {'A':3, 'B':2, 't':30, 'n':2}
res_3 = {'A':3, 'B':3, 't':22, 'n':3}
res = {'res_1':res_1, 'res_2':res_2, 'res_3':res_3}
profit_A = 3000
profit_B = 2500
```

Après avoir formulé le problème avec une structure de données appropriée, essayez de le visualiser en utilisant les fonctions graphiques de Python. Portez le produit A en abscisse, et ne connaissant pas la solution, représentez la quantité à produire du produit A par un vecteur des quantités comprises entre 0 et 30 (les quantités ne pouvant pas être négatives). Pour le produit B (comme dans les formulations qui précèdent), déduisez la quantité à produire des ressources de production qui restent une fois A produit. Formulez trois fonctions, une pour chaque phase, afin de fixer la quantité pour A et d'en déduire la quantité de B, en tenant compte des contraintes.

```
a = np.linspace(0, 30, 30)
c1 = ((res['res_1']['t'] * res['res_1']['n']) -
      res['res_1']['A']*a) / res['res_1']['B']
c2 = ((res['res_2']['t'] * res['res_2']['n']) -
      res['res_2']['A']*a) / res['res_2']['B']
c3 = ((res['res_3']['t'] * res['res_3']['n']) -
```

```

    res['res_3']['A']*a) / res['res_3']['B']

plt.plot(a, c1, label='constrain #1')
plt.plot(a, c2, label='constrain #2')
plt.plot(a, c3, label='constrain #3')

axes = plt.gca()
axes.set_xlim([0,30])
axes.set_ylim([0,30])
plt.xlabel('qty model A')
plt.ylabel('qty model B')

border = np.array((c1,c2,c3)).min(axis=0)

plt.fill_between(a, border, color='yellow', alpha=0.5)
plt.scatter(*zip(*[(0,0), (20,0),
                    (0,20), (16,6), (6,16)]))

plt.legend()
plt.show()

```

Les contraintes sont représentées par trois droites sur le graphique ([Figure 19-2](#)). Leurs intersections déterminent la zone d'acceptabilité. Il s'agit de la zone délimitée par les trois droites, avec des valeurs de A et de B toujours inférieures ou égales aux valeurs situées sur les droites des contraintes (les contraintes sont une frontière au-delà de laquelle ni A ni B ne peuvent prendre leurs valeurs respectives).

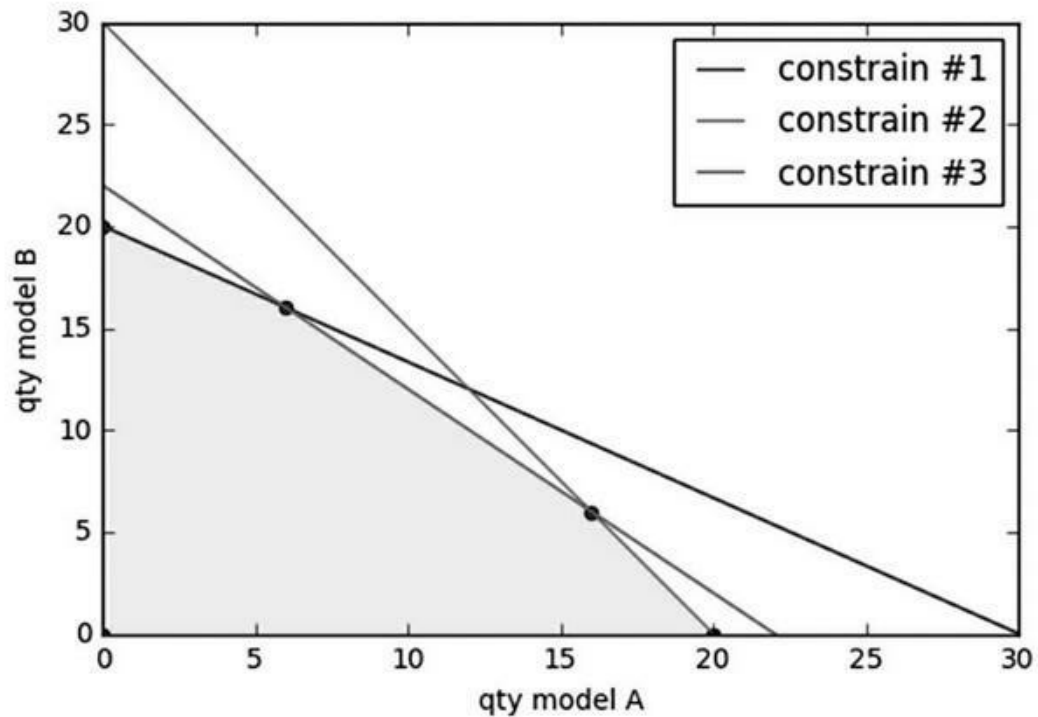


FIGURE 19-2 Quand on cherche quel sommet est la bonne solution.

Selon la méthode du simplexe, la solution optimale correspond à un des cinq sommets du polygone, ces cinq sommets étant (0,0), (20,0), (0,20), (16,6) et (6,16). Pour savoir lequel est la solution, utilisez les fonctions du module PuLP qui sont nécessaires. Tout d'abord, définissez le problème sous forme de modèle. Il s'agit d'un problème de maximisation, A et B devant être positifs.

```
model = pulp.LpProblem("Max profit", pulp.LpMaximize)
A = pulp.LpVariable('A', lowBound=0)
B = pulp.LpVariable('B', lowBound=0)
```



Le solveur de PuLP peut aussi chercher des solutions entières, ce que ne peut pas faire le simplexe dans sa version originale. Ajoutez `cat='Integer'` comme paramètre associé à la définition d'une variable : `A = pulp.LpVariable('A', lowBound=0, cat='Integer')`, ainsi la solution ne comportera que des nombres entiers. Sachez cependant que dans certains problèmes, des résultats sous forme de nombres entiers peuvent se révéler moins optimaux que des résultats sous forme de nombres décimaux : par conséquent,

n'optez pour une solution sous forme de nombres entiers que si cela se justifie (ainsi, par exemple, il n'est pas possible de produire une fraction d'un produit).

Ensuite, ajoutez la fonction objectif, constituée de la somme des deux variables définies par `pulp.LpVariable` et représentant les quantités idéales des produits A et B multipliées par les profits unitaires respectifs.

```
model += profit_A * A + profit_B * B
```

Enfin, ajoutez les contraintes, en procédant exactement de la même manière que pour la fonction objectif. Créez la formulation en utilisant les valeurs appropriées (tirées du dictionnaire de données) et les variables A et B prédéfinies.

```
model += res['res_1']['A'] * A + res['res_1']['B'] * B <= res['res_1']['t'] * res['res_1']['n']
model += res['res_2']['A'] * A + res['res_2']['B'] * B <= res['res_2']['t'] * res['res_2']['n']
model += res['res_3']['A'] * A + res['res_3']['B'] * B <= res['res_3']['t'] * res['res_3']['n']
```

Le modèle est prêt à être optimisé (il intègre la fonction objectif et les contraintes). Appelez la fonction `solve` puis vérifiez son état (il arrive qu'il soit impossible de trouver une solution, ou que la solution trouvée ne soit pas optimale).

```
model.solve()
print ('Completion status: %s'
      % pulp.LpStatus[model.status])
```

```
Completion status: Optimal
```

Après avoir reçu la confirmation que l'optimiseur a trouvé la solution optimale, faites apparaître les quantités relatives des produits A et B.

```
print ("Production du modèle A = %0.1f" % A.varValue)
print ("Production du modèle B = %0.1f" % B.varValue)
```

```
Production du modèle A = 16.0
```

Production du modèle B = 6.0

Faites apparaître aussi le profit total que cette solution permet de réaliser :

```
print ('Profit maximum atteint : %0.1f'  
      % pulp.value(model.objective))
```

Profit maximum atteint : 63000.0

Chapitre 20

À la découverte de l'heuristique

DANS CE CHAPITRE

- » **Savoir quand les heuristiques servent les algorithmes**
 - » **Constater que la recherche de chemin peut être difficile pour un robot**
 - » **Se familiariser rapidement avec l'algorithme de recherche *best-first***
 - » **Améliorer l'algorithme de Dijkstra et prendre le meilleur chemin heuristique avec A***
-

En guise de conclusion, ce chapitre complète l'aperçu de l'heuristique entamé au [Chapitre 18](#) en présentant l'heuristique comme un moyen efficace d'utiliser une recherche locale pour parcourir les solutions voisines. Le [Chapitre 18](#) définit l'heuristique comme une estimation éclairée d'une solution : les heuristiques sont des séries de règles pratiques relatives au résultat désiré, qui permettent aux algorithmes de suivre la bonne voie pour y aboutir. Cependant, une heuristique à elle seule ne peut pas indiquer la manière d'aboutir à la solution.

Il existe des nuances en matière d'heuristique, comme en matière de vérité. Aujourd'hui, les heuristiques sont aux frontières du développement algorithmique. La révolution de l'intelligence artificielle s'appuie sur les algorithmes présentés jusqu'ici dans ce livre et qui ordonnent, organisent, cherchent et manipulent les données entrées. Au sommet de la hiérarchie se trouvent les algorithmes heuristiques utilisés pour l'optimisation, ainsi que les recherches qui déterminent la façon dont les machines apprennent

des données et deviennent capables de résoudre des problèmes en l'absence de toute intervention directe.

Les heuristiques ne sont pas des solutions miracles : il n'existe pas une solution qui résoudrait n'importe quel problème. Les algorithmes heuristiques présentent de sérieux désavantages, et il importe de savoir dans quels cas leur utilisation se justifie. En outre, les heuristiques peuvent conduire les ordinateurs comme les humains à des conclusions fausses. Concernant les humains, des biais qui permettent de gagner du temps quand il s'agit d'évaluer une personne ou une situation peuvent souvent se révéler faux, et même des règles de conduite tirées de l'expérience ne permettent d'aboutir à la bonne solution que dans certaines circonstances. Prenons par exemple l'habitude de donner un coup à un appareil électrique qui ne fonctionne pas. Si le problème provient d'un faux contact, cette méthode peut être couronnée de succès : elle permet parfois de rétablir la connexion électrique. Pour autant, on ne saurait en faire une heuristique générale, car dans d'autres cas cette « solution » peut tout aussi bien s'avérer inefficace, ou même, causer de sérieux dégâts à l'appareil.

Jouer avec les heuristiques

Le mot *heuristique* vient du grec ancien *heuriskein*, qui signifie inventer ou découvrir. Sa signification originale souligne le fait que son emploi soit un moyen pratique de trouver une solution qui n'est pas bien définie, par une recherche et une perception intuitive de la direction à suivre. L'heuristique repose sur la chance et sur une approche par essais et erreurs consistant à essayer différentes solutions. Un *algorithme heuristique*, c'est-à-dire un algorithme fondé sur l'heuristique, résout un problème plus rapidement et plus efficacement du point de vue des ressources informatiques en renonçant à la précision de la solution et à sa complétude, alors que la plupart des algorithmes présentent certaines garanties de résultat. Quand un problème devient trop complexe, l'algorithme heuristique est parfois le seul moyen d'obtenir une solution.

Cerner les objectifs de l'heuristique

L'heuristique peut être le moyen de rendre plus rapides les recherches qui sont longues et exhaustives lorsqu'elles sont effectuées au moyen d'autres solutions, surtout dans le cas des problèmes NP-difficiles dont la résolution implique un nombre exponentiel de tentatives en fonction du nombre d'inputs. Prenons par exemple le problème du voyageur de commerce, ou le problème SAT dans une de ses variantes comme le MAX-3SAT (ces deux problèmes sont évoqués au [Chapitre 18](#)). Les heuristiques déterminent la direction que doit suivre la recherche grâce à une estimation, ce qui élimine un grand nombre de combinaisons qu'autrement l'algorithme aurait dû tester.

Une heuristique, étant une estimation ou une supposition, peut guider l'algorithme qui repose sur elle vers une conclusion fautive, une solution inexacte ou simplement une solution sous-optimale, c'est-à-dire une solution qui serait viable mais qui serait loin d'être la meilleure possible. Dans le cas d'une estimation numérique, par exemple, une heuristique pourrait indiquer que la solution est 41 au lieu de 42. D'autres problèmes souvent associés aux heuristiques sont l'impossibilité de trouver toutes les meilleures solutions et la variabilité du temps d'exécution et des calculs nécessaires pour aboutir à une solution.

L'heuristique est parfaitement adaptée là où d'autres techniques algorithmiques impliqueraient un coût plus élevé. Ainsi, certains problèmes ne peuvent pas être résolus sans heuristique en raison de la mauvaise qualité des données d'entrée et de leur nombre astronomique. Le problème du voyageur de commerce en fait partie : quand vous devez visiter un grand nombre de villes, vous ne pouvez utiliser aucune méthode exacte. Le problème du voyageur de commerce et d'autres problèmes n'admettent aucune solution exacte. Les applications de l'intelligence artificielle relèvent de cette catégorie, sachant que dans ce domaine, de nombreux problèmes comme la reconnaissance de mots parlés ou du contenu d'une image ne peuvent pas être résolus par une séquence exacte d'étapes et de règles.

De la génétique à l'intelligence artificielle

L'étude de la recherche locale du [Chapitre 18](#) présente des méthodes heuristiques comme le recuit simulé et la recherche tabou, qui contribuent à l'optimisation de l'escalade (pour ne pas rester bloqué dans des solutions qui ne sont pas optimales). La famille des heuristiques comprend un certain nombre d'autres applications, notamment :

- » **L'intelligence distribuée** : Un ensemble d'heuristiques fondées sur l'étude du comportement des essaims et des colonies d'insectes (abeilles, fourmis ou lucioles) ou des particules. Cette méthode consiste à procéder par tentatives multiples pour trouver une solution en utilisant des agents (notamment en exécutant plusieurs instances du même algorithme) qui interagissent en coopération les uns avec les autres et avec la problématique. Marco Dorigo, un des meilleurs spécialistes de l'étude des algorithmes d'intelligence distribuée, nous donne davantage de détails sur ce sujet, à l'adresse <http://www.aco-metaheuristic.org/>.
- » **La métaheuristique** : Une heuristique qui vous permet de déterminer (ou même de générer) la bonne heuristique pour votre problème. Parmi les métaheuristiques, les plus répandues sont les *algorithmes génétiques*, inspirés par l'évolution. Un algorithme génétique prend pour point de départ un ensemble de solutions possibles et génère de nouvelles solutions en recourant à la mutation (en ajoutant ou en retirant quelque chose à la solution) et à l'enjambement (consistant à recombinaison différentes parties des solutions lorsqu'une solution est divisible). Dans le problème des n dames ([Chapitre 18](#)), par exemple, on constate que l'on peut diviser un échiquier verticalement car les dames ne se déplacent pas horizontalement, si bien que le problème se prête à l'enjambement. Quand l'ensemble de solutions est suffisamment vaste, les

algorithmes génétiques sélectionnent les solutions en éliminant celles qui ne sont pas viables ou qui ne tiennent pas leurs promesses. Les solutions sélectionnées sont ensuite soumises à une nouvelle itération (mutation, enjambement et sélection). Au bout d'un temps donné et d'un certain nombre d'itérations, les algorithmes génétiques trouvent des solutions meilleures et qui sont très différentes des solutions initiales.

- » **L'apprentissage machine** : Il s'agit de méthodes comme les systèmes neuro-flous, les machines à vecteurs de support et les réseaux neuronaux, qui sont le fondement de l'apprentissage de la machine dans le domaine de l'estimation et du classement à partir d'exemples. À l'image de la façon dont l'enfant apprend par l'expérience, les algorithmes d'apprentissage machine déterminent la manière de fournir la réponse la plus plausible sans appliquer des règles de conduite précises et détaillées (à propos de l'apprentissage machine, lire *Machine Learning For Dummies*, par John Paul Mueller et Luca Massaron, éditions Wiley).
- » **Le routage heuristique** : Il s'agit d'une série d'heuristiques permettant aux robots (mais la méthode est aussi utilisée dans les télécommunications et dans la logistique des transports) de choisir le meilleur chemin pour éviter les obstacles.

Diriger des robots grâce à des heuristiques

Guider un robot dans un environnement inconnu consiste à éviter les obstacles et à atteindre une cible spécifique. Dans le domaine de l'intelligence artificielle, c'est une tâche à la fois fondamentale et difficile. Pour la navigation, les robots peuvent utiliser le *télémètre laser*, la *téledétection par laser* (ou *lidar*, un système

permettant de déterminer grâce à un rayon laser la distance à laquelle se trouve un objet) et le *sonar* (un système qui visualise les environs par le son). Cependant, même équipés des matériels informatiques les plus perfectionnés, les robots ont toujours besoin d'algorithmes appropriés :

- » pour trouver le plus court chemin (ou du moins, un chemin raisonnablement court) vers une destination ;
- » pour éviter les obstacles qui peuvent se trouver sur le passage ;
- » pour adopter des comportements personnalisés, comme tourner ou freiner le moins possible.

Un algorithme de *recherche de chemin* (ou de *planification de trajectoire*) permet à un robot de partir d'un certain lieu et d'atteindre un objectif en utilisant le plus court chemin, en anticipant et en évitant les obstacles (réagir après avoir heurté un mur n'est pas satisfaisant). La recherche de chemin est aussi utile pour déplacer d'autres types d'appareils vers une cible dans l'espace, même une cible virtuelle comme dans un jeu vidéo ou sur des pages Internet.



Le routage autonome est la caractéristique essentielle des voitures sans conducteur et autres véhicules capables de se représenter l'environnement routier et de se rendre à destination sans aucune intervention humaine (il faut cependant dire au véhicule où il doit aller, car il ne lit pas dans les pensées). Voici un article récent du *Guardian* qui donne un bon aperçu des progrès réalisés et des attentes concernant les véhicules sans chauffeur : <https://www.theguardian.com/technology/2015/sep/13/self-driving-cars-bmw-google-2020-driving>.

Explorer des territoires inconnus

Les algorithmes de recherche de chemin exécutent toutes les tâches étudiées précédemment pour trouver le chemin le plus

court, éviter les obstacles, *etc.* Ils peuvent utiliser pour cela deux sortes de cartes schématiques de l'environnement :

- » **Les cartes topologiques** : Ce sont des schémas simplifiés dont sont absents tous les détails superflus. On y trouve les principaux points de repère, les sens de circulation et des indications de distances. Parmi les exemples du monde réel, on peut citer les plans de métro de Tokyo (<http://www.tokyometro.jp/en/subwaymap/>) et de Londres (<https://tfl.gov.uk/maps/track/tube>).
- » **Les grilles d'occupation** : Le territoire y est divisé en petits carrés ou en petits hexagones, qui sont remplis quand les capteurs du robot détectent un obstacle dans la zone qu'ils représentent. L'Université technique de Prague nous en propose un exemple : <http://cmp.felk.cvut.cz/cmp/demos/Omni/mobil/>. Vous pouvez aussi regarder les vidéos suivantes qui montrent comment un robot assemble et visualise une carte : <https://www.youtube.com/watch?v=zjl7NmutMlc> et <https://www.youtube.com/watch?v=RhPlzIyTT58>.

Vous pouvez visualiser les cartes topologiques et les grilles d'occupation en tant que représentations graphiques. Cependant, un algorithme a surtout besoin d'une structure de données appropriée. En l'occurrence, la meilleure structure de données est le graphe, sachant que les sommets peuvent facilement représenter des carrés, des hexagones ou des repères de balisage. Les arêtes relient les sommets de la même manière que les routes, les chemins ou les passages relient des points géographiques.



Votre appareil de navigation GPS fonctionne en utilisant des graphes. Les cartes routières en couleurs que l'appareil affiche sur l'écran, avec leur défilement continu et tous leurs détails, sont constituées à partir d'ensembles de sommets et d'arêtes parcourus par des algorithmes qui vous permettent de trouver votre chemin tout en évitant les bouchons.

La représentation du territoire du robot sous forme de graphe nous renvoie aux problèmes étudiés au [Chapitre 9](#), consacré au parcours d'un sommet à un autre par le plus court chemin. Le plus court chemin peut être celui qui passe par le plus petit nombre de sommets, ou bien celui qui représente le moindre coût (calculé comme la somme des poids des arêtes parcourues, ces poids pouvant représenter des distances ou une autre grandeur). En voiture, vous tenez compte non pas uniquement de la distance à parcourir pour atteindre votre destination, mais également de certaines conditions comme la circulation (certains itinéraires pouvant être encombrés, voire impraticables), l'état des routes et les limites de vitesse.

Lorsqu'il s'agit de trouver le plus court chemin vers une destination dans un graphe, les algorithmes les plus simples et les plus évidents selon la théorie des graphes sont la recherche en profondeur et l'algorithme de Dijkstra (présenté au [Chapitre 9](#)). La recherche en profondeur consiste à explorer le graphe en s'éloignant le plus possible du point de départ, puis en revenant aux étapes précédentes pour explorer d'autres chemins jusqu'à ce que la destination soit trouvée. L'algorithme de Dijkstra explore le graphe de façon judicieuse et selon une méthode gloutonne, en ne tenant compte que des chemins les plus courts. En dépit de leur simplicité, ces algorithmes sont très efficaces quand il s'agit d'évaluer les différents chemins possibles dans un graphe simple et d'en obtenir une vue globale, avec une connaissance complète des directions à suivre pour atteindre la destination et moyennant un coût réduit.

Dans le cas d'un robot, la situation est légèrement différente car le robot ne peut pas percevoir tous les chemins possibles à la fois, étant limité en termes de visibilité (les obstacles peuvent occulter le chemin, ou bien la cible peut être trop éloignée). Le robot découvre son environnement à mesure qu'il se déplace, et dans le meilleur des cas, il peut évaluer la distance et la direction de sa destination finale. Tout se passe comme dans un labyrinthe, plus exactement, comme dans un labyrinthe végétal où l'on arrive à s'orienter et à repérer sa destination à distance.



Il existe des haies végétales un peu partout dans le monde. Les plus fameuses ont été créées en Europe entre le milieu du XVI^e siècle et le XVIII^e siècle. Dans un labyrinthe végétal, c'est-à-dire constitué de haies, la hauteur des haies ne permet pas de voir où l'on va. On peut cependant s'orienter (lorsqu'il est possible de voir le soleil), et même repérer l'endroit où se trouve la sortie (voir par exemple <https://www.venetoinside.com/hidden-treasures/post/maze-of-villa-pisani-in-stravenice/>). On retrouve aussi de fameux labyrinthes végétaux dans des films comme *Shining* de Stanley Kubrick, ou *Harry Potter et la Coupe de feu*.

Utiliser des mesures de distance comme heuristiques

Quand un problème du monde réel ne peut pas être résolu par une méthode algorithmique précise parce que l'input est ambigu, manquant ou instable, les heuristiques peuvent être utiles. Lorsqu'il s'agit de trouver un chemin en se fondant sur des coordonnées dans un plan cartésien (la cartographie plane utilisant un ensemble de coordonnées horizontales et verticales), deux simples mesures donnent la distance entre deux points sur un plan : la distance euclidienne et la distance de Manhattan.

On utilise communément la distance euclidienne car elle procède du théorème de Pythagore sur les triangles. Pour connaître la distance en ligne droite entre deux points A et B d'un plan, dont on connaît les coordonnées, on peut les considérer comme les points extrêmes de l'hypoténuse (le côté le plus long d'un triangle). Comme l'indique la [Figure 20-1](#), on calcule la distance à partir de la longueur des deux autres côtés en utilisant un troisième point, le point C, de même coordonnée horizontale que B et de même coordonnée verticale que A.

Il s'agit de calculer les différences respectives entre les coordonnées horizontales et verticales des deux points, de les élever au carré (pour qu'elles deviennent positives), de les

additionner, et enfin, d'utiliser la racine carrée du résultat. Dans cet exemple, la distance entre A et B est calculée à partir de leurs coordonnées (1 ; 2) et (3 ; 3) :

$$\text{sqrt}((1-3)^2 + (2-3)^2) = \text{sqrt}(2^2+1^2) = \text{sqrt}(5) = 2.236$$

La distance de Manhattan se calcule d'une autre manière. On commence par additionner les longueurs des côtés B et C, ce qui revient à additionner les valeurs absolues des différences entre les coordonnées horizontales et verticales des points A et B :

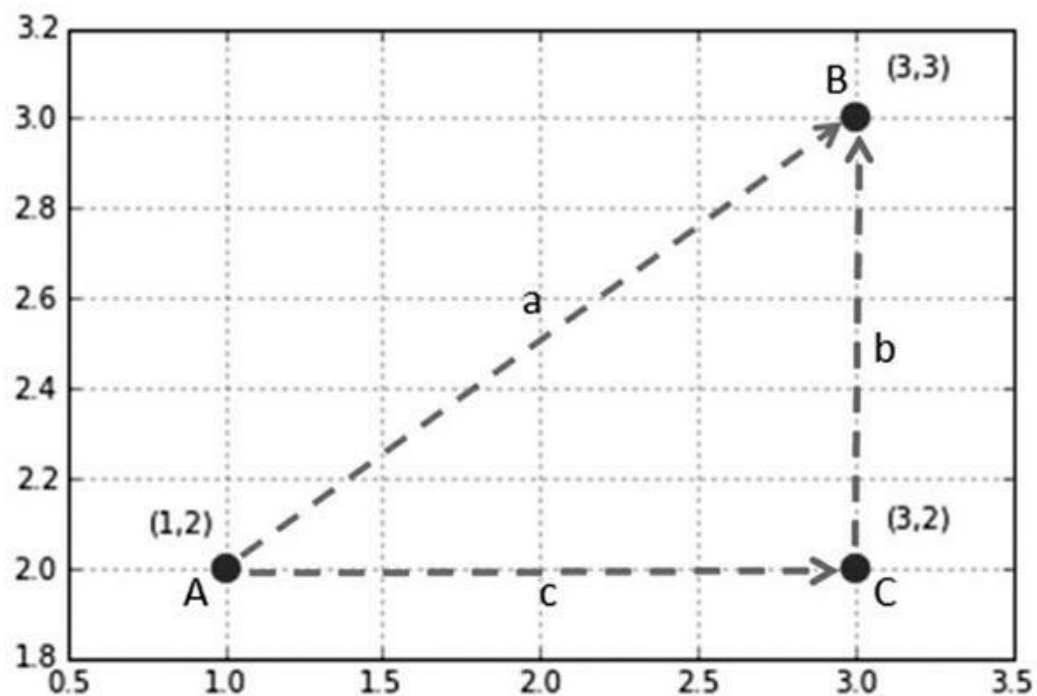


FIGURE 20-1 A et B sont deux points sur une carte, dont on connaît les coordonnées.

$$|(1-3)| + |(2-3)| = 2 + 1 = 3$$

La distance euclidienne indique le plus court chemin, tandis que la distance de Manhattan donne le chemin le plus long mais le plus plausible quand des obstacles sont à prévoir sur la trajectoire la plus directe. Le chemin en question représente la trajectoire d'un taxi dans Manhattan (d'où ce nom), le taxi longeant les pâtés de

maisons pour atteindre sa destination (une ligne droite entre les deux points ne serait pas envisageable, car elle traverserait des bâtiments). Cette distance est parfois aussi appelée distance city-block. Par conséquent, si vous devez vous déplacer d'un point A à un point B sans savoir si vous rencontrerez des obstacles, faire un détour par le point C est une bonne heuristique car c'est la distance à prévoir dans le cas le moins favorable.

Expliquer les algorithmes de recherche de chemin

Cette dernière partie du chapitre est consacrée à expliquer deux algorithmes fondés sur l'heuristique, l'algorithme de recherche best-first et l'algorithme A* (prononcer « A star »). Les sections qui suivent montrent que ces deux algorithmes donnent une solution rapide à un problème de labyrinthe associé à une carte topologique ou à une grille d'occupation représentée sous forme de graphe. Ces deux algorithmes sont couramment utilisés dans les domaines de la robotique et des jeux vidéo.

Créer un labyrinthe

Comme mentionné précédemment, la carte topologique et la grille d'occupation ressemblent à un labyrinthe végétal, surtout s'il existe des obstacles entre le point de départ et le point d'arrivée. Des algorithmes spécialisés permettent de créer et de traiter des labyrinthes, plus particulièrement le parcours en profondeur (connu depuis l'Antiquité : vous faites courir votre main sur un mur du labyrinthe et vous ne l'en retirez jamais tant que vous n'avez pas gagné la sortie) et l'algorithme de Pledge (pour plus de détails sur les sept classifications des labyrinthes, consultez la page <http://www.astrolog.org/labyrnth/algrithm.htm>). Cependant, la recherche de chemin est fondamentalement différente de la résolution des problèmes de labyrinthe car dans le premier cas, on

sait où se trouve l'objectif tandis que dans le second cas, il faut résoudre le problème en ignorant où se trouve la sortie.

En conséquence, la procédure pour simuler un labyrinthe parsemé d'obstacles qu'un robot doit traverser suit une approche différente et plus simple. Plutôt qu'un jeu d'obstacles, on détermine un graphe dont les sommets sont répartis dans une grille (à l'image d'une carte) et l'on supprime des connexions de façon aléatoire pour simuler la présence d'obstacles. Le graphe n'est pas orienté (chaque arête peut être parcourue dans un sens comme dans l'autre), mais il est pondéré, car le parcours d'un sommet à un autre dure un temps donné. En particulier, un mouvement en diagonale demande plus de temps qu'un mouvement de haut en bas ou de gauche à droite. La première étape consiste à importer les modules Python nécessaires. Ensuite, le code définit les fonctions de distance euclidienne et de Manhattan (vous retrouverez ce code dans le fichier téléchargeable A4D ; 20 ; Heuristic Algorithms.ipynb sur le site *Dummies* : pour plus de détails, consultez l'Introduction).

```
import numpy as np
import string
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

def euclidean_dist(a, b, coord):
    (x1, y1) = coord[a]
    (x2, y2) = coord[b]
    return np.sqrt((x1-x2)**2+(y1-y2)**2)

def manhattan_dist(a, b, coord):
    (x1, y1) = coord[a]
    (x2, y2) = coord[b]
    return abs(x1 - x2) + abs(y1 - y2)

def non_informative(a,b):
    return 0
```

L'étape suivante consiste à créer une fonction pour produire des labyrinthes de façon aléatoire. Le point de départ est un nombre entier de votre choix. Le labyrinthe généré sera toujours le même

lorsque vous saisirez le même nombre. Autrement, le processus est entièrement aléatoire.

```
def create_maze(seed=2, drawing=True):
    np.random.seed(seed)
    letters = [l for l in string.ascii_uppercase[:25]]
    checkboard = np.array(letters[:25]).reshape((5,5))
    Graph = nx.Graph()
    for j, node in enumerate(letters):
        Graph.add_nodes_from(node)
        x, y = j // 5, j % 5
        x_min = max(0, x-1)
        x_max = min(4, x+1)+1
        y_min = max(0, y-1)
        y_max = min(4, y+1)+1
        adjacent_nodes = np.ravel(
            checkboard[x_min:x_max,y_min:y_max])
        exits = np.random.choice(adjacent_nodes,
            size=np.random.randint(1,4), replace=False)
        for exit in exits:
            if exit not in Graph.edge[node]:
                Graph.add_edge(node, exit)
        spacing = np.arange(0.0, 1.0, 0.2)
        coordinates = [[x,y] for x in spacing \
            for y in spacing]
        position = {l:c for l,c in zip(letters,
            coordinates)}

    for node in Graph.edge:
        for exit in Graph.edge[node]:
            length = int(round(
                euclidean_dist(
                    node, exit, position)*10,0))
            Graph.add_edge(node,exit,weight=length)

    if drawing:
        nx.draw(Graph, position, with_labels=True)
        labels = nx.get_edge_attributes(Graph, 'weight')
        nx.draw_networkx_edge_labels(Graph, position,
            edge_labels=labels)
    plt.show()

    return Graph, position
```

Les fonctions retournent un graphe NetworkX (Graph), une structure de données de prédilection pour la représentation des

graphes, qui comporte 25 sommets (ou nœuds, si vous préférez) et une représentation cartésienne des points (de leurs positions). Les sommets sont disposés sur une grille de 5 x 5 ([Figure 20-2](#)). L'output tient compte aussi des fonctions de distance et calcule la position des sommets.

```
graph, coordinates = create_maze(seed=3)
```

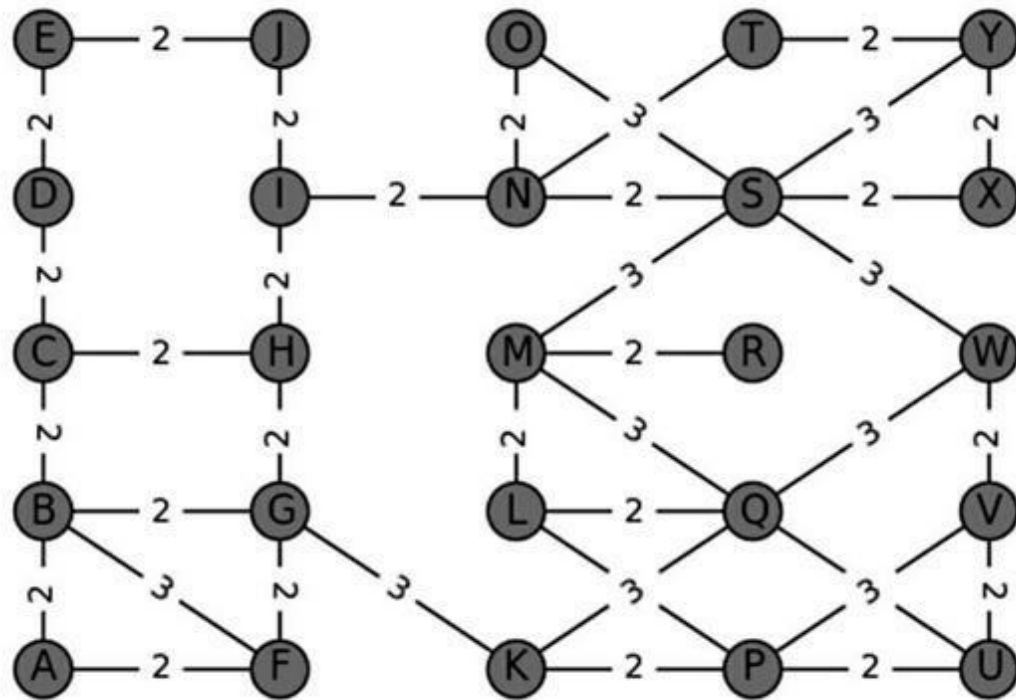


FIGURE 20-2 Un labyrinthe représentant une carte topologique avec des obstacles.



Dans le labyrinthe généré par une valeur de départ de 2, chacun des sommets est relié aux autres. Le processus de génération étant aléatoire, certaines cartes comportent des sommets non connectés, entre lesquels il n'y a donc pas de trajet possible. Pour voir comment cela fonctionne, essayez une valeur de départ de 13. Cette situation peut se produire dans la réalité : il arrive, par exemple, qu'un robot ne puisse pas atteindre une certaine destination.

Chercher un chemin rapide par la méthode du best-first

L'algorithme de recherche en profondeur explore le graphe en passant d'un sommet à l'autre et en ajoutant des directions à une structure de données en pile. Quand le moment est venu d'effectuer un mouvement, l'algorithme emprunte la première direction indiquée par la pile. Tout se passe comme si l'on traversait un labyrinthe comportant de nombreuses pièces en se dirigeant toujours vers la première sortie que l'on aperçoit. Le plus probable est l'arrivée finale à un cul-de-sac qui n'est pas la destination. Il faut ensuite revenir sur ses pas, traverser à nouveau les pièces précédemment visitées en cherchant une autre sortie, mais cela demande beaucoup de temps quand on est éloigné de la destination.

L'heuristique peut être très utile ici, avec la répétition qu'entraîne la recherche en profondeur. Le recours à l'heuristique permet de savoir si l'on se rapproche ou si l'on s'éloigne de la cible. La combinaison ainsi utilisée est ce que l'on appelle l'algorithme de recherche *best-first* (*Best-first search algorithm*, ou BFS) . Ce nom comporte le mot « best » parce qu'en explorant le graphe, le programme ne retient pas la première arête qu'il trouve, mais procède à une évaluation et choisit la solution qui, selon l'heuristique, devrait nous rapprocher du résultat désiré. La méthode fait penser à l'optimisation gloutonne, et cet algorithme est parfois appelé *recherche gloutonne best-first*. Le BFS n'atteint généralement pas la cible à la première tentative, mais grâce à l'heuristique, le parcours se termine à son voisinage et il y a moins de retours en arrière qu'en utilisant uniquement la recherche en profondeur.



L'algorithme BFS est utilisé principalement par les robots qui recherchent certaines informations sur le Web. Il permet à un agent logiciel de parcourir un graphe globalement inconnu en recourant à l'heuristique pour évaluer le degré de proximité entre le contenu de la prochaine page et celui de la page initiale (en vue

de rechercher le meilleur contenu). Cet algorithme est aussi largement utilisé dans le domaine des jeux vidéo pour déplacer les personnages contrôlés par l'ordinateur lorsqu'ils cherchent des ennemis ou des récompenses. Il ressemble donc à un algorithme glouton à action ciblée.

La démonstration d'un BFS dans Python à l'aide du labyrinthe précédemment créé illustre la façon dont un robot peut se déplacer dans un espace en le percevant comme un graphe. Le code suivant fait intervenir deux fonctions générales qui sont aussi utilisées par le prochain algorithme présenté dans cette section. Ces deux fonctions indiquent les directions à suivre à partir d'un sommet (`node_neighbors`) et déterminent le coût du trajet d'un sommet à un autre (`graph_weight`). Le poids représente la distance ou le temps.

```
def graph_weight(graph, a, b):
    return graph.edge[a][b]['weight']

def node_neighbors(graph, node):
    return graph.edge[node]
```

L'algorithme de planification de trajectoire simule le déplacement d'un robot dans un graphe. Quand il trouve une solution, la planification se traduit par un déplacement. Les algorithmes de planification de trajectoire produisent donc un résultat qui indique le meilleur parcours entre deux sommets, mais il faut toujours une fonction pour traduire l'information, déterminer l'itinéraire à suivre et en calculer la longueur. Les fonctions `reconstruct_path` et `compute_path` produisent le plan sous forme d'étapes et d'espérance de coût à partir du résultat de l'algorithme de planification de trajectoire.

```
def reconstruct_path(connections, start, goal):
    if goal in connections:
        current = goal
        path = [current]
        while current != start:
            current = connections[current]
            path.append(current)
        return path[::-1]
```

```
def compute_path_dist(path, graph):
    if path:
        run = 0
        for step in range(len(path)-1):
            A = path[step]
            B = path[step+1]
            run += graph_weight(graph, A, B)
        return run
    else:
        return 0
```

Une fois qu'il a préparé toutes les fonctions de base, le programme crée un labyrinthe en utilisant une valeur de départ de 30. Dans ce labyrinthe, on distingue deux principaux itinéraires entre un point A et un point Y, sachant qu'il existe des obstacles au milieu de la carte ([Figure 20-3](#)). Il y a aussi des impasses sur le parcours (comme les sommets E et O).

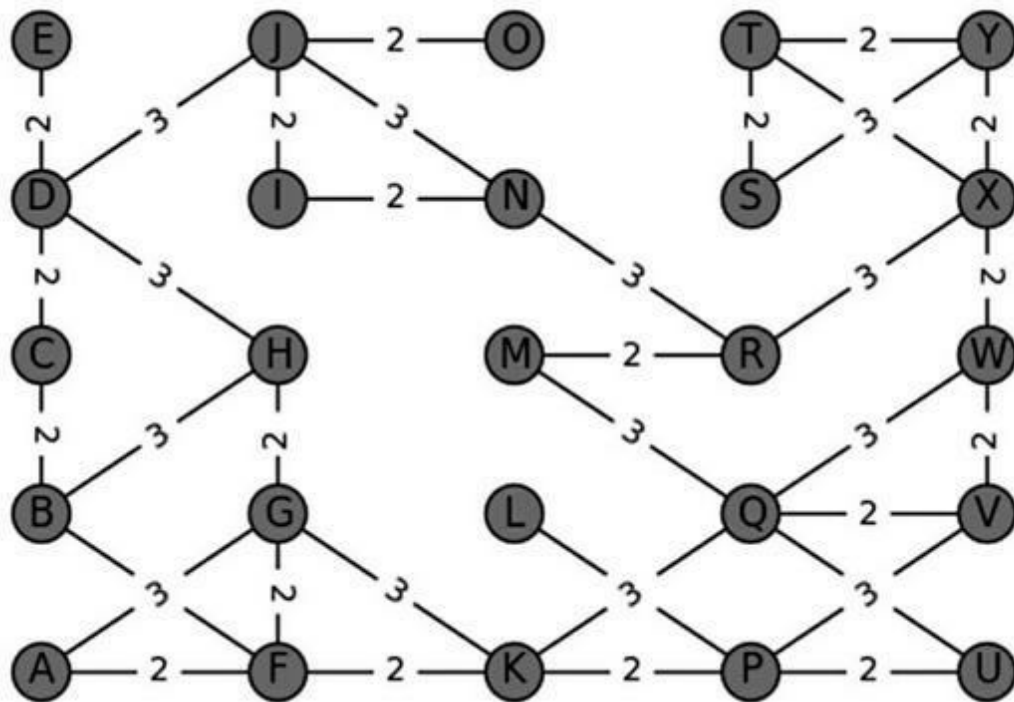


FIGURE 20-3 Un labyrinthe compliqué doit être résolu en recourant à l'heuristique.

```
graph, coordinates = create_maze(seed=30)
```



```

start = 'A'
goal = 'Y'
scoring=manhattan_dist

```

L'implémentation du BFS est un peu plus complexe que le code de la recherche en profondeur présenté au [Chapitre 9](#). Deux listes sont utilisées : une pour les sommets jamais visités (appelée `open_list`), une autre pour les sommets visités (`closed_list`). La liste `open_list` sert de file de priorité : une règle de priorité détermine le premier élément à extraire. En l'occurrence, l'heuristique définit la priorité, si bien que la file de priorité fournit une direction qui nous rapproche de la cible. L'heuristique de la distance de Manhattan est la plus efficace en raison des obstacles qui encombrant le chemin vers la destination :

```

# Best-first search
path = {}
open_list = set(graph.nodes())
closed_list = {start: manhattan_dist(start, goal,
                                     coordinates)}
while open_list:

    candidates = open_list&closed_list.keys()
    if len(candidates)==0:
        print ("Pas de chemin trouvé vers l'objectif %s"
% goal)
        break
    frontier = [(closed_list[node],
                 node) for node in candidates]
    score, min_node =sorted(frontier)[0]
    if min_node==goal:
        print ("Arrivée au sommet final %s" % goal)
        print ('Sommets non visités : %i' % (len(
            open_list)-1))
        break
    else:
        print("Traitement du sommet %s, " % min_node,
end="")

    open_list = open_list.difference(min_node)
    neighbors = node_neighbors(graph, min_node)
    to_be_visited = list(neighbors-closed_list.keys())

    if len(to_be_visited) == 0:

```

```

        print ("pas de sortie trouvée, retour à %s"
              % path[min_node])
    else:
        print ("découvert %s" % str(to_be_visited))

    for node in neighbors:
        if node not in closed_list:
            closed_list[node] = scoring(node, goal,
                                         coordinates)
            path[node] = min_node

print ('\nMeilleur chemin :', reconstruct_path(
    path, start, goal))
print ('Longueur de ce chemin : %i' %
      compute_path_dist(
        reconstruct_path(path, start, goal), graph))

Traitement du sommet A, découvert ['F', 'G']
Traitement du sommet G, découvert ['K', 'H']
Traitement du sommet H, découvert ['B', 'D']
Traitement du sommet D, découvert ['E', 'J', 'C']
Traitement du sommet J, découvert ['O', 'I', 'N']
Traitement du sommet O, pas de sortie trouvée, retour
à J
Traitement du sommet N, découvert ['R']
Traitement du sommet R, découvert ['M', 'X']
Traitement du sommet X, découvert ['T', 'W', 'Y']
Arrivée au sommet final Y
Sommets non visités : 15

Meilleur chemin : ['A', 'G', 'H', 'D', 'J', 'N', 'R',
                  'X',
                  'Y']
Longueur de ce chemin : 22

```

Dans cet exemple, le résultat est parlant. On peut voir comment fonctionne l'algorithme. Le BFS poursuit sa recherche jusqu'à ce qu'il n'y ait plus de sommets à explorer. S'il est passé par tous les sommets sans avoir atteint la cible, le code vous indique qu'il ne peut pas atteindre la cible et le robot ne se déplacera pas. Si le programme trouve la destination, il cesse de traiter les sommets, même si `open_list` en contient encore : ainsi, on économise du temps.

Quand le programme rencontre une impasse, par exemple au sommet O, il recherche un chemin précédemment ignoré. La meilleure alternative se présente immédiatement grâce à la file de priorité, et l'algorithme la saisit. Dans cet exemple, le BFS évite de perdre du temps sur 15 sommets, suit le chemin ascendant sur la carte et effectue un parcours de A à Y en 22 étapes.



Vous pouvez essayer d'autres labyrinthes en fixant un nombre de départ différent et en comparant les résultats du BFS avec ceux de l'algorithme A* présenté dans la prochaine section. Vous constaterez que le BFS choisit parfois le meilleur chemin de façon rapide et efficace, mais pas toujours. Si vous voulez un robot capable d'effectuer une recherche rapide, le BFS est le meilleur choix.

Effectuer un parcours heuristiquement avec A*

L'algorithme A* détermine rapidement le chemin le plus court dans un graphe en associant la recherche gloutonne de Dijkstra étudiée au [Chapitre 9](#) à une technique d'arrêt précoce (il s'arrête dès qu'il a atteint le sommet de destination) et à une estimation heuristique (généralement basée sur la distance de Manhattan) indiquant la partie du graphe à explorer en premier. A* a été mis au point en 1968 au Centre d'intelligence artificielle du Stanford Research Institute (aujourd'hui appelé SRI International) dans le cadre du projet du robot Shakey. Shakey a été le premier robot mobile capable de décider de façon autonome comment se rendre quelque part (cependant son champ d'exploration se limitait aux quelques salles du laboratoire). Afin de rendre Shakey entièrement autonome, ses développeurs avaient mis au point l'algorithme A*, la transformée de Hough (une technique de traitement d'image pour la reconnaissance des formes d'un objet), et la méthode du graphe de visibilité (une technique de représentation de chemin sous forme de graphe). L'article de la page <http://www.ai.sri.com/shakey/> décrit Shakey plus en détail, et le

montre même en action. Il reste surprenant aujourd'hui de voir, sur cette vidéo, ce que Shakey était capable de faire : <https://www.youtube.com/watch?v=qXdn6ynwpil>. L'algorithme A^* est actuellement le meilleur algorithme pour la recherche du meilleur chemin dans un graphe lorsque les informations et les prévisions dont on dispose sont incomplètes (comme le reflète la fonction heuristique qui guide la recherche). A^* est capable :

- » **de trouver à chaque fois le plus court chemin**, à condition que ce chemin existe et que l'algorithme A^* soit correctement informé par l'estimation heuristique. A^* repose sur l'algorithme de Dijkstra, qui garantit l'aboutissement à la meilleure solution.
- » **de trouver la solution plus vite que n'importe quel autre algorithme**, à condition de disposer d'une heuristique juste, qui indique les bonnes directions pour atteindre le voisinage de la cible de façon similaire, mais plus judicieuse encore, à l'algorithme BFS.
- » **de calculer les poids lors du parcours des arêtes**. Les poids tiennent compte du coût du déplacement dans une certaine direction. Ainsi, par exemple, effectuer un virage peut demander plus de temps qu'aller tout droit, comme dans l'exemple du robot Shakey.



Une heuristique correcte et *admissible* donne une information utile à l'algorithme A^* sur la distance à la cible, en ne surestimant jamais le coût de son accès. Par ailleurs, A^* recourt à son heuristique davantage que BFS, par conséquent l'heuristique doit effectuer les calculs rapidement, faute de quoi le processus global de traitement serait trop long.

Dans cet exemple, Python utilise le même code et les mêmes structures de données que pour le BFS, mais il existe des différences. Les principales différences sont que l'algorithme, au cours de son déroulement, met à jour le coût du trajet entre le sommet de départ et chacun des sommets explorés. En outre, pour choisir un itinéraire, A^* prend en compte le plus court chemin

entre le point de départ et la cible en passant par le sommet courant, sachant qu'il additionne l'estimation issue de l'heuristique et le coût du chemin calculé jusqu'au sommet courant. Ce processus permet à l'algorithme d'effectuer plus de calculs que le BFS quand l'heuristique constitue une estimation correcte, et de déterminer le meilleur chemin possible.



Trouver le plus court chemin possible en termes de coût est la fonction fondamentale de l'algorithme de Dijkstra. A* est simplement un algorithme de Dijkstra dans lequel le coût pour atteindre un sommet est calculé de façon améliorée grâce à l'heuristique de la distance escomptée jusqu'à la cible. Le [Chapitre 9](#) présente en détail l'algorithme de Dijkstra. En vous reportant à l'étude en question, vous pourrez mieux saisir le fonctionnement de l'algorithme A* et la façon dont il tire parti de l'heuristique.

```
# A*
open_list = set(graph.nodes())
closed_list = {start: manhattan_dist(
    start, goal, coordinates)}
visited = {start: 0}
path = {}

while open_list:

    candidates = open_list&closed_list.keys()
    if len(candidates)==0:
        print ("Pas de chemin trouvé vers l'objectif %s"
% goal)
        break
    frontier = [(closed_list[node],
        node) for node in candidates]
    score, min_node =sorted(frontier)[0]

    if min_node==goal:
        print ("Arrivée au sommet final : %s" % goal)
        print ('Sommet non visités : %i' % (len(
            open_list)-1))
        break
    else:
        print("Traitement du sommet %s, " % min_node,
end="")
```

```

open_list = open_list.difference(min_node)
current_weight = visited[min_node]
neighbors = node_neighbors(graph, min_node)
to_be_visited = list(neighbors-visited.keys())

for node in neighbors:
    new_weight = current_weight + graph_weight(
        graph, min_node, node)
    if node not in visited or \
    new_weight < visited[node]:
        visited[node] = new_weight
        closed_list[node] = manhattan_dist(node,
goal,
        coordinates) + new_weight
        path[node] = min_node

    if to_be_visited:
        print ("découvert %s" % to_be_visited)
    else:
        print ("retour à open list")
print ('\nMeilleur chemin :', reconstruct_path(
    path, start, goal))
print ('Longueur de ce chemin : %i' %
compute_path_dist(
    reconstruct_path(path, start, goal), graph))

```

```

Traitement du sommet A, découvert ['F', 'G']
Traitement du sommet F, découvert ['B', 'K']
Traitement du sommet G, découvert ['H']
Traitement du sommet K, découvert ['Q', 'P']
Traitement du sommet H, découvert ['D']
Traitement du sommet B, découvert ['C']
Traitement du sommet P, découvert ['L', 'U', 'V']
Traitement du sommet Q, découvert ['M', 'W']
Traitement du sommet C, retour à open list
Traitement du sommet U, retour à open list
Traitement du sommet D, découvert ['E', 'J']
Traitement du sommet V, retour à open list
Traitement du sommet L, retour à open list
Traitement du sommet W, découvert ['X']
Traitement du sommet E, retour à open list
Traitement du sommet M, découvert ['R']
Traitement du sommet J, découvert ['O', 'I', 'N']
Traitement du sommet X, découvert ['T', 'Y']
Traitement du sommet R, retour à open list
Traitement du sommet O, retour à open list
Traitement du sommet I, retour à open list
Arrivée au sommet final Y

```

Sommets non visités : 3

Meilleur chemin : ['A', 'F', 'K', 'Q', 'W', 'X', 'Y']

Longueur de ce chemin : 14

Lorsque l'algorithme A* a terminé l'analyse du labyrinthe, il indique un chemin optimal qui est bien plus court que celui donné par l'algorithme BFS. Cette solution a son revers : A* explore presque tous les sommets présents, n'en laissant de côté que trois. Comme dans le cas de l'algorithme de Dijkstra, le temps d'exécution dans le pire des cas est $O(v^2)$, où v est le nombre de sommets dans le graphe, ou bien $O(e + v * \log(v))$, où e est le nombre d'arêtes, quand on utilise des files de priorité : une structure de données efficiente lorsqu'il s'agit d'obtenir la valeur minimum dans une longue liste. Du point de vue du temps d'exécution le plus défavorable, l'algorithme A* n'est pas différent de celui de Dijkstra, quoiqu'il soit plus performant en moyenne sur les graphes de grande dimension, car il trouve en premier le sommet cible lorsqu'il est guidé correctement par l'évaluation heuristique (dans le cas du robot, la distance de Manhattan).

PARTIE 6

La Partie des Dix

DANS CETTE PARTIE...

Constater la façon dont les algorithmes changent le monde

Découvrir l'avenir des algorithmes

Définir des problèmes que les algorithmes n'ont pas résolus

Apprendre comment des jeux permettent de résoudre des problèmes algorithmiques

Chapitre 21

Dix algorithmes qui sont en train de changer le monde

DANS CE CHAPITRE

- » Étudier des routines de tri et de recherche
 - » Utiliser des nombres au hasard
 - » Réduire les données
 - » Assurer la confidentialité des données, et plus encore...
-

O n imagine difficilement qu'un algorithme puisse révolutionner un domaine ou un autre, et plus difficilement encore qu'il puisse changer le monde. Et cependant, aujourd'hui les algorithmes sont partout et vous ne réalisez sans doute pas quel impact ils peuvent avoir sur votre quotidien.

Beaucoup de gens se rendent compte que les boutiques en ligne et autres plateformes commerciales sur Internet utilisent des algorithmes pour déterminer les produits complémentaires à suggérer aux acheteurs en fonction de leurs achats précédents. En revanche, les gens ignorent le plus souvent que des algorithmes sont utilisés en médecine, notamment comme aide au diagnostic.

Les algorithmes font leur apparition là où l'on s'y serait attendu le moins. Le minutage des feux de signalisation est souvent déterminé par des calculs effectués par des algorithmes. Ce sont des algorithmes qui permettent à votre smartphone de vous parler,

et c'est grâce à des algorithmes que votre téléviseur peut vous offrir des fonctionnalités qui n'avaient jamais existé auparavant sur ce type d'appareil. On peut donc penser que les algorithmes sont en voie de changer le monde. Ce chapitre vous en présente dix.



Les puristes pourront dire que les algorithmes ont toujours changé le monde et qu'il n'y a donc rien de nouveau de ce côté depuis des siècles et des siècles. Déjà vers 1600 av. J.-C., les Babyloniens utilisaient des algorithmes pour effectuer des factorisations et calculer des racines carrées. Al-Khawarizmi a décrit des algorithmes de résolution d'équations linéaires et quadratiques vers l'an 820. Ce chapitre est consacré à des applications informatiques de l'algorithmique, mais les algorithmes sont apparus il y a fort longtemps.

Utiliser des routines de tri

Sans le tri des données, plus grand-chose ne serait possible. Pour pouvoir exploiter des données, encore faut-il pouvoir y accéder. Vous trouverez des centaines d'algorithmes de tri sur des sites comme <https://betterexplained.com/articles/sorting-algorithms/> et ce livre vous en présente plusieurs (voir [Chapitre 7](#)).

Cependant, les trois routines de tri les plus couramment utilisées sont Mergesort, Quicksort et Heapsort, car elles sont particulièrement rapides (pour une comparaison des temps d'exécution, voir

<http://www.cprogramming.com/tutorial/computersciencetheory/sorting.html>

Quelle sera la routine de tri la plus indiquée pour votre application ? Tout dépend des facteurs suivants :

- » ce que vous voulez obtenir avec votre application ;
- » le type de données que vous utilisez ;
- » les ressources informatiques dont vous disposez.

Le fait est que la capacité de trier les données pour qu'elles se présentent sous la forme exigée par une application, afin qu'une certaine tâche puisse être exécutée, est déterminante : elle conditionne toute l'activité mondiale.

De nos jours, certaines entreprises doivent tout aux algorithmes de tri. Prenons l'exemple de Google : l'activité du moteur de recherche repose pour une grande part sur la capacité de trier les données pour les rendre rapidement accessibles. Imaginons aussi la difficulté que nous aurions à trouver un article sur le site d'Amazon s'il n'y avait pas une routine de tri. Même certaines applications installées sur votre ordinateur sont basées sur le tri des données. Il n'est sans doute pas exagéré de dire que toute application informatique est très dépendante d'une routine de tri.

Des programmes et des sous-programmes de recherche

À l'instar des routines de tri, les routines de recherche font partie de pratiquement n'importe quelle application informatique aujourd'hui, quelle qu'en soit la taille. Ces applications sont partout, même là où on pourrait ne pas s'attendre à les trouver, comme dans votre voiture. Trouver rapidement l'information est essentiel dans notre quotidien. Imaginons, par exemple, que vous soyez en retard à votre rendez-vous et que vous vous rendiez compte que votre GPS ne trouve pas l'adresse. Comme les routines de tri, les routines de recherche se présentent sous différentes formes et ont différentes tailles, et elles sont décrites sur des sites comme

<https://tekmarathon.com/2012/10/05/bestsearching-algorithm-2/>

ou

<http://research.cs.queensu.ca/home/cisc121/2006s/webnotes/search>

En fait, les routines de recherche sont plus courantes encore que les routines de tri car les exigences de recherche sont souvent plus prégnantes et plus complexes. Un certain nombre de routines de

recherche sont d'ailleurs présentées dans ce livre (voir [Chapitre 7](#)).

Faire bouger les choses avec les nombres au hasard

Sans le hasard, toutes sortes de choses seraient moins amusantes. Qui aimerait jouer au solitaire ou aux sept familles si tout était déterminé ? La génération de nombres au hasard est un élément essentiel de l'expérience de jeu. Comme on l'a vu dans plusieurs chapitres qui précèdent, un certain degré d'aléa est nécessaire au bon fonctionnement de certains algorithmes (voir, par exemple, la section du [Chapitre 15](#) intitulée « Organiser des données en cache »). Dans certains cas, les essais sont plus probants quand on utilise des valeurs aléatoires (voir, par exemple, la section du [Chapitre 14](#) intitulée « Choisir un type particulier de compression »).



Les nombres au hasard générés par un algorithme sont en réalité pseudo-aléatoires, ce qui signifie qu'il devient possible de prédire le prochain nombre d'une série quand on connaît l'algorithme et la valeur de départ utilisée. C'est pourquoi cette information est si farouchement protégée.



Les nombres pseudo-aléatoires générés par des algorithmes ne sont pas nécessaires à toutes les applications ni à toutes les machines (mais néanmoins à une vaste majorité d'entre elles). Il existe des méthodes de génération des nombres au hasard basées sur le bruit atmosphérique ou sur les variations de température (pour plus de détails, voir <http://engineering.mit.edu/ask/can-computergenerate-truly-random-number>). En fait, une solution matérielle comme ChaosKey (<http://altusmetrum.org/ChaosKey/>), que vous pouvez brancher sur votre port USB, produit vraisemblablement de véritables nombres au hasard. Le site de ChaosKey indique par un schéma de quelle manière ce système recueille le bruit aléatoire pour le transformer en nombres au

hasard.

Pratiquer la compression des données

Le [Chapitre 14](#) étudie les techniques de compression généralement utilisées pour les fichiers. La compression des données concerne aujourd'hui tous les aspects de l'informatique. L'utilisation des images, des vidéos et des fichiers audio, par exemple, passe par la compression de données. Sans compression de données, il serait impossible d'obtenir le niveau de rendement exigé par certaines tâches comme la lecture de films en ligne en continu.

Cependant, la compression des données trouve davantage encore d'applications qu'on pourrait l'imaginer. Elle est utilisée par tous les systèmes de gestion de base de données (SGBD), de telle sorte que les données n'occupent sur le disque dur qu'une quantité d'espace raisonnable. L'informatique en nuage ne fonctionnerait pas sans compression de données : le temps de téléchargement des fichiers entre le nuage et les machines locales serait trop long. Même les pages Web dépendent souvent de la compression des données, lorsque des informations doivent être transférées d'un endroit à un autre.

Garder les données secrètes

L'idée de garder des données secrètes n'est pas nouvelle. C'est même l'une des plus anciennes raisons d'utiliser un algorithme. Le mot cryptographie vient de deux mots grecs : *kryptós* (caché, ou secret) et *graphein* (écriture). Les Grecs ont probablement été les premiers à utiliser la cryptographie, et des textes datant de l'Antiquité indiquent que Jules César communiquait avec ses généraux au moyen de missives cryptées. Quoi qu'il en soit, le secret des données est une des plus longues batailles de l'histoire. Dès que quelqu'un invente une nouvelle technique pour tenir des informations secrètes, quelqu'un d'autre trouve un moyen de

casser le code et de percer le secret. Aujourd'hui, la cryptographie assistée par l'ordinateur englobe les éléments suivants :

- » **La confidentialité** : Faire en sorte que personne ne puisse accéder aux informations échangées entre deux parties.
- » **L'intégrité des données** : Réduire le risque que le contenu des données transmises entre deux parties puisse être modifié.
- » **L'authentification** : Déterminer l'identité d'une ou plusieurs parties.
- » **La non-répudiation** : Limiter la possibilité qu'une des parties puisse remettre en cause un engagement pris.



L'histoire des algorithmes de cryptage informatique est longue et intéressante. Une liste des algorithmes communément utilisés (aujourd'hui et dans le passé) peut être consultée sur les pages <http://www.cryptographyworld.com/algo.htm> et <https://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/crypto.html>. Le guide consultable à l'adresse https://www.owasp.org/index.php/Guide_to_Cryptography donne des détails supplémentaires sur les principes de fonctionnement de la cryptographie.

Changer le domaine des données

La transformée de Fourier et la transformation de Fourier rapide (FFT) ont un impact considérable sur la façon dont les applications reçoivent les données. Ces deux algorithmes font passer les données du domaine des fréquences (vitesse d'oscillation du signal) au domaine temporel (différentiel de temps entre les modifications du signal). En fait, il est impossible de prétendre maîtriser l'informatique sans avoir longuement étudié ces deux algorithmes. Le facteur temps est essentiel.

Connaître la fréquence à laquelle quelque chose change permet d'évaluer l'intervalle de temps entre deux changements, et donc



de savoir pendant combien de temps on doit exécuter une tâche avant qu'un changement d'état entraîne un besoin différent. Ces algorithmes sont couramment utilisés pour différentes sortes de filtrages. Sans ce filtrage, il serait impossible de reproduire fidèlement des séquences audio ou vidéo *via* une connexion en continu. Toutes ces applications peuvent paraître assez sophistiquées, et elles le sont, mais vous trouverez des tutoriaux remarquables qui vous donneront une meilleure idée du fonctionnement de ces algorithmes (voir par exemple le tutoriel sur la page <http://w.astro.berkeley.edu/~jrg/ngst/fft/fft.html>). Le tutoriel de la page <https://betterexplained.com/articles/aninteractive-guide-to-the-fourier-transform/> est probablement le plus intéressant et il est particulièrement divertissant.

Analyser les liens

La capacité d'analyser les liens est une des propriétés les plus remarquables de l'informatique d'aujourd'hui. La Troisième partie de ce livre était consacrée à la représentation de ces liens sous forme graphique, préalablement à l'analyse. En fait, l'Internet repose avant tout sur la création de liens, et la connectivité était bien une idée prédominante dans les débuts de ce qui est devenu un phénomène mondial. Sans la capacité d'analyser et d'exploiter des liens, des applications comme les bases de données et les messageries électroniques ne pourraient pas exister : vous ne pourriez certainement pas communiquer avec vos amis sur Facebook.

À mesure que le Web s'est développé et que les gens se sont familiarisés aux appareils et aux systèmes qui rendaient la connectivité plus simple et omniprésente, des applications comme Facebook et des sites de vente comme Amazon ont recouru davantage à l'analyse de liens, notamment pour vous vendre davantage de produits. Bien sûr, cette connectivité a aussi son côté négatif (voir par exemple <http://www.pcmag.com/commentary/351623/facebook-a-toolfor->

[evil](#)), mais pour l'essentiel, c'est l'analyse de liens qui nous permet d'être mieux informés et davantage en contact avec le monde qui nous entoure.

Naturellement, l'analyse des liens ne se limite pas à l'information sous forme connectée. Elle peut servir, par exemple, à fournir des indications d'itinéraire aux automobilistes ou à déterminer des liens de causalité entre les activités humaines et les maladies. Elle nous permet de percevoir les liens entre des facteurs que nous aurions tendance à négliger, mais qui exercent un véritable impact sur notre vie quotidienne. Grâce à l'analyse des liens, votre médecin peut vous conseiller des changements à entreprendre dans vos habitudes pour prévenir certains risques et vivre plus longtemps. Partout, les choses sont liées. L'analyse de liens nous fournit une méthode pour identifier ces relations et savoir lesquelles sont réellement importantes.

Repérer les schémas de données

Les données n'existent pas dans l'absolu. Elles dépendent de toutes sortes de facteurs, y compris des préjugés et autres partis pris qui influencent notre perception. Le [Chapitre 10](#) explique comment les données ont tendance à se regrouper dans certains environnements et montre que l'analyse de ces regroupements peut être riche d'enseignements.



L'analyse des tendances et des structures joue un rôle de premier plan dans certaines applications actuelles de l'informatique parmi les plus remarquables. La méthode de détection d'objets de Viola et Jones, par exemple, rend possible la reconnaissance des visages en temps réel. Il s'agit d'un algorithme susceptible de permettre d'améliorer la sécurité dans des lieux publics comme les aéroports, où s'affairent aujourd'hui des individus malfaisants. Des algorithmes similaires pourraient permettre aux médecins de détecter diverses sortes de cancers bien avant que ces cancers deviennent visibles à l'œil nu, et une détection précoce accroît la probabilité de rémission complète. Il en est de même pour toutes

sortes de problèmes médicaux (comme la détection des fractures qui peuvent être trop petites pour être visibles, mais néanmoins douloureuses).

La reconnaissance des formes sert aussi des fins plus banales. Ainsi, par exemple, ce type d'analyse permet aux automobilistes de détecter les problèmes de circulation avant de s'y trouver confrontés. L'analyse des structures peut aussi permettre aux agriculteurs de produire davantage et à moindre coût en utilisant l'eau et les engrais de façon plus rationnelle. La reconnaissance des formes peut aussi guider des drones dans le survol des cultures. Ainsi les agriculteurs peuvent gérer leur temps plus efficacement et travailler davantage de surface moyennant un coût moins élevé. Sans les algorithmes, tout cela ne serait pas possible.

Gérer l'automatisation et les réponses automatiques

L'algorithme de régulation proportionnelle intégrale dérivée, c'est tout un programme. Essayez donc de le dire trois fois plus vite ! Il s'agit d'un des algorithmes secrets les plus importants. Vous n'en avez probablement jamais entendu parler, et pourtant il vous sert tous les jours. Cet algorithme particulier est celui d'un mécanisme de contrôle d'asservissement qui minimise la marge d'erreur entre le signal de sortie désiré et le signal de sortie réel. Il est utilisé partout pour contrôler les automatismes. Quand votre voiture commence à glisser parce que vous freinez trop fort, par exemple, c'est grâce à cet algorithme que le système automatique antiblocage, le fameux ABS, joue son rôle. Sans cela, l'ABS risquerait de surcompenser et d'aggraver la situation.

Aujourd'hui, l'algorithme de régulation proportionnelle intégrale dérivée est utilisé pratiquement partout où l'on utilise une machine. Sans cet algorithme, la robotique serait impossible. Imaginez ce qui se produirait dans une usine si tous les robots surcompensaient systématiquement, quelle que soit leur activité :

le chaos qui en résulterait inciterait rapidement les responsables à renoncer à utiliser des machines, pour quelque usage que ce soit.

Créer des identifiants uniques

Tout se passe comme si chacun de nous était un numéro : en fait, non pas un numéro seulement, mais un certain nombre de numéros. Notre carte bancaire est identifiée par un numéro, tout comme notre permis de conduire et notre carte d'identité, et il en est de même de toutes sortes d'entreprises et d'organisations. Nous en arrivons à devoir tenir à jour une liste de tous ces numéros, car il y en a trop pour que nous puissions les connaître tous par cœur. Et cependant, chacun de ces numéros est censé identifier une personne de façon unique aux yeux d'un de ses interlocuteurs. Derrière toute cette unicité, il y a diverses sortes d'algorithmes.

Le [Chapitre 7](#) traite notamment du hachage, qui est un moyen d'assurer cette unicité. Le hachage et la cryptographie reposent l'un et l'autre sur la décomposition en produits de facteurs premiers, un type d'algorithme qui décompose les très grands nombres en nombres premiers. La décomposition en produits de facteurs premiers fait partie des problèmes les plus difficiles à résoudre avec les algorithmes, mais elle fait l'objet de recherches incessantes. La société actuelle est tellement dépendante de notre capacité de nous identifier de façon univoque que les secrets de la création de ces identifiants y jouent un rôle essentiel.

Chapitre 22

Dix problèmes d'algorithmique non encore résolus

DANS CE CHAPITRE

- » Effectuer facilement des recherches dans les textes
 - » Détecter les différences entre les mots
 - » Envisager la faisabilité de systèmes d'hypercalculs
 - » Utiliser des fonctions à sens unique, et plus encore...
-

Sachant que cela fait des siècles que l'on utilise des algorithmes, on pourrait penser que les scientifiques ont déjà découvert tous les algorithmes et ont déjà résolu tous les problèmes d'algorithmique imaginables. Malheureusement, il n'en est rien, bien au contraire. Souvent, la résolution d'un problème d'algorithmique soulève davantage de questions que l'algorithme ne résout pas, et cela ne se voit pas tant que quelqu'un n'a pas trouvé la solution. En outre, le progrès des technologies et l'évolution des modes de vie font souvent apparaître de nouveaux défis, et pour y répondre, il faut mettre au point de nouveaux algorithmes. C'est le cas, par exemple, de l'univers de plus en plus connecté dans lequel nous nous retrouvons plongés aujourd'hui et de l'utilisation de plus en plus courante des robots.

Comme expliqué au [Chapitre 1](#), un algorithme est une série d'étapes servant à résoudre un problème et il convient de ne pas confondre les algorithmes avec, par exemple, les équations. Un

algorithme n'est jamais une solution à la recherche d'un problème. Personne ne mettrait au point une série d'étapes pour résoudre un problème qui n'existe pas encore (ou qui pourrait ne jamais exister). Par ailleurs, de nombreux problèmes sont intéressants sans qu'il existe un besoin pressant de leur trouver une solution. Par conséquent, même si tout le monde est averti d'un tel problème et comprend que certains pourraient souhaiter qu'une solution soit trouvée, personne ne s'empressera de s'y atteler.

Ce chapitre est consacré aux problèmes d'algorithmique dont la solution, si elle était trouvée un jour, serait utile. En résumé, si ce chapitre n'est pas à négliger, c'est parce que vous pourriez rencontrer un problème que vous aimeriez vraiment voir résolu, et vous pourriez même avoir la motivation de faire partie de ceux qui le résoudraient.

Trouver des expressions pour la recherche dans les textes

La recherche textuelle passe souvent par l'utilisation d'expressions régulières, d'une sorte d'abrégié qui indique à l'ordinateur ce qu'il doit trouver. La grammaire utilisée dépend du langage utilisé ou de l'application, mais on trouve des expressions régulières dans divers contextes, par exemple dans les logiciels de traitement de texte, les messageries électroniques, les outils de recherche et partout où il est nécessaire de saisir des termes de recherche précis pour trouver un ensemble d'éléments textuels. Pour plus de détails sur les expressions régulières, consultez la page <http://www.regular-expressions.info/>.



Un problème courant avec les expressions régulières est que tout se passe comme si les règles étaient similaires d'une application à une autre, mais avec juste assez de différences pour qu'il soit difficile de mettre au point un terme de recherche. Le problème de hauteur d'étoile généralisée consiste à chercher s'il existe une

syntaxe généralisée pour les expressions régulières. Si oui, l'algorithme résultant rendrait possible l'apprentissage d'une méthode unique pour créer des expressions régulières en vue d'effectuer les recherches. Pour plus de détails sur ce problème, consultez la page <https://www.irif.fr/~jep/Problemes/starheight.html>.

Différencier des mots

Pour l'ordinateur, les caractères que nous utilisons ne sont pas des lettres, mais des nombres. Ces nombres, en réalité, ne sont que des séries de 0 et de 1 sans signification. Quand nous combinons des caractères pour former des chaînes, nous ne faisons finalement qu'allonger les séries de 0 et de 1. Par conséquent, la comparaison entre deux chaînes de caractères, que nous pouvons généralement faire en un instant, est pour l'ordinateur un travail qui demande du temps et qui peut porter à confusion.

Ainsi, par exemple, si l'on n'est pas suffisamment circonspect dans l'élaboration de l'algorithme, l'ordinateur peut confondre *cuvé* et *vécu*. Plus important, l'ordinateur a besoin de temps pour faire la différence entre ces deux mots. Le problème de la séparation des mots consiste à trouver l'algorithme le plus court (et le plus rapide) possible (en l'occurrence, un automate fini déterministe, ou AFD) pour réaliser cette séparation. L'objectif est d'accepter un mot tout en rejetant un autre, parmi deux mots d'une longueur donnée.

Déterminer si un programme finira par s'arrêter

Un des problèmes qu'Alan Turing avait proposés en 1936 était de savoir si un algorithme, compte tenu de la description d'un programme et d'un input, pourrait déterminer si ce programme finira par s'arrêter ou non (le *problème de l'arrêt*). Quand on

utilise une application simple, il est souvent possible de savoir si le programme finira par s'arrêter ou s'il va continuer à mouliner en suivant une boucle sans fin. Cependant, quand un programme est complexe, il devient plus difficile de déterminer le résultat de son exécution pour un input donné. Une machine de Turing ne peut pas le déterminer : le résultat est un code plein de bogues et contenant des boucles infinies. Avec les technologies actuelles, quel que soit le nombre d'essais effectués, il n'est pas possible de résoudre ce problème.



Un *système d'hypercalculs* est un modèle de calcul qui va plus loin que la machine de Turing pour résoudre des problèmes comme le problème de l'arrêt. Cependant, les technologies actuelles ne permettent pas de disposer d'une telle machine. Si une telle machine existait, on pourrait la confronter à toutes sortes d'impondérables auxquels les ordinateurs actuels ne peuvent pas répondre. L'article de la page <https://www.newscientist.com/article/mg22329781-500-what-willhypercomputers-let-us-do-good-question/> constitue un bon aperçu de ce qu'il adviendrait si quelqu'un parvenait à résoudre ce problème.

Créer et utiliser des fonctions à sens unique

Une fonction à sens unique est une fonction qui est facile à utiliser dans un sens pour obtenir une réponse, mais pratiquement impossible à inverser. En d'autres termes, une fonction à sens unique peut servir à créer, par exemple, un hachage dans le cadre d'une solution de cryptographie, d'identification de personnes, d'authentification, ou pour d'autres besoins en matière de sécurité des données.



La question de l'existence d'une fonction à sens unique n'a pas grand-chose de mystérieux, c'est plutôt une question de preuve. Actuellement, de nombreux systèmes de télécommunications, de

commerce électronique et de banque électronique dépendent de fonctions qui sont censées être à sens unique, mais personne ne sait vraiment si elles le sont. Il s'agit d'une hypothèse, et non d'une théorie (à propos de la différence entre hypothèse et théorie, voir http://www.diffen.com/difference/Hypothesis_vs_Theory). Si quelqu'un pouvait prouver qu'il existe une fonction à sens unique, les problèmes de sécurité des données seraient plus faciles à résoudre, du point de vue de la programmation.

Multiplier des nombres vraiment très grands

Des nombres vraiment très grands sont utilisés un peu partout. Prenons l'exemple des calculs de distance de la Terre à la planète Mars, ou bien à Pluton. Lorsque les nombres sont trop grands pour pouvoir être stockés dans les registres du processeur, leur multiplication doit être effectuée en plusieurs étapes. Les calculs font alors intervenir un certain nombre d'opérations, si bien qu'ils prennent beaucoup plus de temps. Les méthodes de calcul actuelles sont les suivantes :

- » l'algorithme de multiplication des nombres complexes de Gauss ;
- » l'algorithme de Karatsuba ;
- » l'algorithme Toom-Cook ;
- » les méthodes de transformation de Fourier.

Les méthodes actuelles produisent souvent des résultats acceptables, mais aucune n'est rapide. Or, quand on a beaucoup de calculs à effectuer, le temps peut devenir un problème critique. C'est pourquoi la multiplication des grands nombres fait partie des problèmes exigeant une solution meilleure que celles dont on dispose aujourd'hui.

Partager équitablement des ressources

Diviser des ressources de manière égale ne semble pas difficile, mais beaucoup de gens envieux trouveront la répartition des ressources inéquitable tant qu'on n'aura pas trouvé un moyen de les assurer du contraire. C'est le même problème que couper un gâteau sans que personne ne se sente floué. Bien sûr, cette situation est inévitable, même avec la meilleure volonté du monde. Or, dans toute organisation, l'allocation équitable des ressources est importante au quotidien pour minimiser les conflits entre les parties prenantes et pour rendre les gens plus efficaces.



Pour résoudre le problème du gâteau à couper en un nombre donné de parts égales, il existe déjà deux solutions, mais aucune solution générale. S'il n'y a que deux personnes, la première coupera le gâteau et la seconde choisira sa part. Le problème se complique quand il faut partager le gâteau en trois, mais la solution de Selfridge-Conway vous est proposée sur la page <https://ochronus.com/cutting-the-pie> (que ce soit pour un gâteau ou pour une tarte, la procédure est la même). Cependant, à partir de quatre parts il n'existe aucune solution.

Réduire le temps de calcul de la distance d'édition

La *distance d'édition* entre deux chaînes de caractères est le nombre d'opérations nécessaires pour passer d'une chaîne à l'autre. Le calcul de la distance comporte les opérations de Levenshtein, c'est-à-dire la suppression, l'insertion et la substitution d'un caractère de la chaîne. Cette technique est appliquée aux interfaces en langage naturel, à la quantification des séquences d'ADN et à toutes sortes d'autres situations dans

lesquelles deux chaînes similaires doivent être comparées ou modifiées.

Il existe déjà pour ce problème un certain nombre de solutions, mais qui ne sont pas rapides du tout. En effet, leur complexité est généralement exponentielle, le temps d'exécution des transformations successives augmente rapidement, si bien que des temps morts se remarquent dans le traitement des données d'entrée. Ce n'est pas un grand mal quand on utilise un traitement de texte qui contrôle automatiquement les mots et corrige les fautes d'orthographe. En revanche, avec les interfaces vocales, les temps morts deviennent gênants et conduisent l'utilisateur à commettre des erreurs. L'objectif actuel est de parvenir à un calcul de distance d'édition en temps sous-quadratique : $O(n^{2-\epsilon})$.

Résoudre des problèmes rapidement



Avec l'essor de l'apprentissage machine et le recours de plus en plus généralisé aux ordinateurs pour la résolution des problèmes, la rapidité du traitement informatique devient une question cruciale. Le problème $P = NP$ consiste simplement si un ordinateur peut résoudre un problème rapidement quand il peut en vérifier rapidement la solution. En d'autres termes, si l'ordinateur peut raisonnablement vérifier dans un temps polynomial (ou plus rapidement) qu'une réponse donnée par l'utilisateur à un problème est correcte, peut-il également résoudre ce problème dans un temps polynomial (ou plus rapidement) ?

Cette question avait été initialement étudiée dans les années cinquante par John Nash dans des courriers adressés à la National Security Agency (NSA), puis à nouveau dans des lettres échangées entre Kurt Gödel et John von Neumann. Outre l'apprentissage machine (et l'intelligence artificielle de façon plus générale), ce problème particulier intéresse un certain nombre de disciplines comme les mathématiques, la cryptographie, la recherche algorithmique, la théorie des jeux, le traitement multimédia, la philosophie et l'économie.

Jouer au jeu de parité

Au premier abord, la résolution d'un problème de jeu peut sembler ne pas être d'une grande utilité dans la vie pratique. Certes, les jeux peuvent être amusants et intéressants, mais ils ne débouchent sur rien de bien utile, du moins le pense-t-on généralement. Pourtant, la théorie des jeux intervient dans un grand nombre de situations réelles, souvent dans des processus élaborés qui peuvent être compris plus facilement quand ils sont envisagés comme des jeux. C'est le cas notamment pour la vérification automatisée et la synthèse de contrôleurs. Pour plus de détails sur le jeu de parité, consultez la page <http://www.sciencedirect.com/science/article/pii/S0890540115000> Vous pouvez même y jouer, sur la page <https://www.abefehr.com/parity/>.

Aborder les questions spatiales

Pour replacer ce problème particulier dans son contexte, prenons l'exemple des caisses à disposer dans un entrepôt, ou envisageons d'autres situations dans lesquelles il importe de gérer l'espace dans lequel des objets évoluent. Bien évidemment, s'il s'agit de stocker un grand nombre de caisses dans un vaste entrepôt en les déplaçant à l'aide d'un chariot élévateur, il ne s'agit pas de tout réorganiser physiquement jusqu'à parvenir à la solution optimale, mais plutôt de visualiser une solution.

Cependant, la question est de savoir si tous les problèmes d'espace ont une solution. Dans les puzzles et autres jeux consistant à assembler une image en retrouvant la bonne disposition des petits éléments qui la constituent, il semble qu'il existe toujours une solution. Cependant, dans certains cas, un mauvais point de départ peut engendrer une situation dans laquelle il n'y a pas de solution. Pour plus de détails concernant ce problème, consultez la page

<http://math.stackexchange.com/questions/754827/does-a-15-puzzle-always-have-a-solution>.



Des mathématiciens comme Sam Loyd (voir <https://www.mathsisfun.com/puzzles/sam-loyd-puzzles-index.html>) utilisent souvent des puzzles pour exposer des problèmes mathématiques complexes, qui n'ont parfois pas de solution connue. La visite de ces sites Internet est intéressante, car on y trouve non seulement de quoi s'amuser, mais aussi matière à réfléchir. Les problèmes que posent ces puzzles et autres énigmes ont véritablement des applications pratiques, même s'ils sont présentés dans un esprit ludique.

Sommaire

[Couverture](#)

[Les algorithmes pour les Nuls](#)

[Copyright](#)

[Introduction](#)

[À propos de ce livre](#)

[Idées reçues](#)

[Icônes utilisées dans ce livre](#)

[Pour aller plus loin](#)

[Par où commencer ?](#)

[PARTIE 1. Pour commencer](#)

[Chapitre 1. Introduction à l'algorithmique](#)

[Décrire les algorithmes](#)

[Utiliser l'informatique pour résoudre des problèmes](#)

[Distinguer les problèmes et les solutions](#)

[Structurer les données pour obtenir une solution](#)

[Chapitre 2. Étude de la conception des algorithmes](#)

[Commencer à résoudre un problème](#)

[Diviser pour régner](#)

[La gloutonnerie n'est pas toujours un vilain défaut](#)

[Calculer les coûts et suivre une heuristique](#)

[Évaluer les algorithmes](#)

[Chapitre 3. Utiliser Python pour faire de l'algorithmique](#)

[Prendre la mesure des avantages de Python](#)

[Découvrir les modules de Python](#)

[Installer Python sous Linux](#)

[Installer Python sous macOS \(<https://fr.wikipedia.org/wiki/MacOS>\)](#)

[Installer Python sous Windows](#)

[Télécharger les jeux de données et le code exemple](#)

[Chapitre 4. Utiliser Python pour la programmation algorithmique](#)

[Travailler avec des nombres et des règles logiques](#)

[Créer et utiliser des chaînes](#)

[Interagir avec des dates](#)

[Créer et utiliser des fonctions](#)

[Utiliser des instructions conditionnelles et des boucles](#)

[Stocker des données à l'aide d'ensembles, de listes et de tuples](#)

[Définir des itérateurs utiles](#)

[Indexer les données à l'aide de dictionnaires](#)

[Chapitre 5. Effectuer des manipulations de données essentielles à l'aide de Python](#)

[Effectuer des calculs avec des vecteurs et des matrices](#)

[Créer des combinaisons de la bonne manière](#)

[Obtenir les résultats désirés grâce à la récursivité](#)

[Exécuter les tâches plus rapidement](#)

[PARTIE 2. L'importance du tri et de la recherche de données](#)

[Chapitre 6. Structurer les données](#)

[Pourquoi les données doivent être structurées](#)

[Empiler les données dans le bon ordre](#)

[Exploiter les structures arborescentes](#)

[Représenter les relations à l'aide d'un graphe](#)

[Chapitre 7. Organiser et rechercher les données](#)

[Trier les données à l'aide du tri fusion et du tri rapide](#)

[Utiliser les arbres de recherche et le tas](#)

[Recourir au hachage](#)

[PARTIE 3. Explorer le monde des graphes](#)

[Chapitre 8. Assimiler les bases de la théorie des graphes](#)

[Apprécier l'importance des réseaux](#)

[Comment tracer un graphe](#)

[Mesurer la fonctionnalité d'un graphe](#)

[Mettre un graphe sous forme numérique](#)

[Chapitre 9. Relier les points](#)

[Parcourir un graphe de manière efficiente](#)

[Trier les éléments du graphe](#)

[Réduire à un arbre couvrant minimum](#)

[Trouver le plus court chemin](#)

[Chapitre 10. Découvrir les secrets des graphes](#)

[Envisager les réseaux sociaux comme des graphes](#)

[Parcourir un graphe](#)

[Chapitre 11. Obtenir la bonne page Web](#)

[Un moteur de recherche pour avoir le monde entier](#)

[Expliquer l'algorithme PageRank](#)

[Mettre en œuvre PageRank](#)

[Au-delà du paradigme de PageRank](#)

[PARTIE 4. Dans l'univers des grandes données](#)

[Chapitre 12. Gérer les grandes données](#)

[Transformer l'énergie électrique en données](#)

[Gérer le flux de données](#)

[Obtenir une réponse des données d'un flux](#)

[Chapitre 13. Effectuer des opérations en parallèle](#)

[Gérer des quantités colossales de données](#)

[Étudier des algorithmes pour MapReduce](#)

[Chapitre 14. Compresser les données](#)

[Rendre les données moins volumineuses](#)

[PARTIE 5. Traiter des problèmes difficiles](#)

[Chapitre 15. Travailler avec des algorithmes gloutons](#)

[Quand faut-il être glouton ?](#)

[Trouver l'utilité qu'il peut y avoir à être glouton](#)

[Chapitre 16. Utiliser la programmation dynamique](#)

[Expliquer la programmation dynamique](#)

[Découvrir les meilleures formules dynamiques](#)

[Chapitre 17. Utiliser des algorithmes probabilistes](#)

[Comment fonctionne la randomisation ?](#)

[Mettre de l'aléa dans votre logique](#)

[Chapitre 18. Effectuer une recherche locale](#)

[La recherche locale, qu'est-ce que c'est ?](#)

[Les trucs de la recherche locale](#)

[Expliquer l'escalade avec les dames des échecs](#)

[Résoudre la satisfiabilité des circuits booléens](#)

[Résoudre 2-SAT grâce à la randomisation](#)

[Chapitre 19. Faire appel à la programmation linéaire](#)

[Utiliser les fonctions linéaires comme outil](#)

[Faire une application pratique de la programmation linéaire](#)

[Chapitre 20. À la découverte de l'heuristique](#)

[Jouer avec les heuristiques](#)

[Diriger des robots grâce à des heuristiques](#)

[Expliquer les algorithmes de recherche de chemin](#)

[PARTIE 6. La Partie des Dix](#)

[Chapitre 21. Dix algorithmes qui sont en train de changer le monde](#)

[Utiliser des routines de tri](#)

[Des programmes et des sous-programmes de recherche](#)

[Faire bouger les choses avec les nombres au hasard](#)

[Pratiquer la compression des données](#)

[Garder les données secrètes](#)

[Changer le domaine des données](#)

[Analyser les liens](#)

[Repérer les schémas de données](#)

[Gérer l'automatisation et les réponses automatiques](#)

[Créer des identifiants uniques](#)

[Chapitre 22. Dix problèmes d'algorithmique non encore résolus](#)

[Trouver des expressions pour la recherche dans les textes](#)

[Différencier des mots](#)

[Déterminer si un programme finira par s'arrêter](#)

[Créer et utiliser des fonctions à sens unique](#)

[Multiplier des nombres vraiment très grands](#)

[Partager équitablement des ressources](#)

[Réduire le temps de calcul de la distance d'édition](#)

[Résoudre des problèmes rapidement](#)

[Jouer au jeu de parité](#)

[Aborder les questions spatiales](#)